UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

19|20

Trabajo fin de grado

ACELERACIÓN DE PROGRAMAS PYTHON MEDIANTE CYTHON

Jonathan Ruiz Gallardo

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

Campus Internacional excelencia UAM CSIC+

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

ACELERACIÓN DE PROGRAMAS PYTHON MEDIANTE CYTHON

Autor: Jonathan Ruiz Gallardo
Tutor: José Ramón Dorronsoro Ibero

junio 2020

Jonathan Ruiz Gallardo
ACELERACIÓN DE PROGRAMAS PYTHON MEDIANTE CYTHON

Jonathan Ruiz Gallardo

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

*A mis **padres**,*
por su enorme esfuerzo y apoyo en este largo viaje.

*A mi **hermano**,*
mi punto de Lagrange.

Поехали!

Ein großer Fehler: daß man sich mehr dünkt, als man ist, und sich weniger schätzt, als man wert ist.

– Johann Wolfgang von Goethe

Agradecimientos

A dos personas en especial del profesorado de la Autónoma, la primera es *Pilar Rodríguez*, una amiga; la segunda, y sin la que este humilde documento nunca hubiera visto la luz, es *José Ramón*, por su enorme paciencia, sus consejos y por darme la oportunidad de finalizar con éxito esta etapa académica. A los dos, gracias.

A mis amigos, en especial a mi *Noe*, por estar ahí siempre; a *Fran* y a *Manu*, que sus visitas y nuestras cervezas en la terraza no tienen precio; a Damjanka, que su risa es un regalo; a *Tanya* y a *Keika*, que nuestro amor por los idiomas nos hizo conocernos; a *Armance*, que la echo de menos.

A *Gregory C. Ewing* y a *Stefan Behnel*, que enriquecieron en sabiduría este Trabajo de Fin de Grado con las entrevistas que me concedieron.

A toda la *comunidad de Cython*, que hace posible esta bella y potente herramienta.

A mis padres y a mi hermano, este trabajo es para ellos; uno de tantos, vendrán más.

Jonathan Ruiz Gallardo,
Berlín, 17.04.2020

Resumen

A pesar de su dinamismo, Python es un lenguaje con prolongados tiempos de ejecución, especialmente en algoritmos que posean una alta carga numérica. Por esta razón, librerías especializadas como h5py, scikit-image o Numpy, entre otras, están desarrolladas de forma parcial o en un alto porcentaje en Cython.

En este TFG se presenta una introducción al lenguaje Cython y se estudian el algoritmo de Bellman-Ford y el de Ford-Fulkerson, en los que se obtienen, para cada uno de ellos, los tiempos de ejecución en tres casos diferentes: con una versión de Python pura, con Python compilado a través de Cython y, finalmente, con una versión íntegra de Cython. Posteriormente, se explican las ecuaciones diferenciales junto a su clasificación y sus métodos para su resolución numérica y se obtienen los tiempos con los tres casos anteriormente citados en el sistema de ecuaciones de Lorenz mediante la utilización del método de Euler.

Los resultados obtenidos en la resolución de ecuaciones diferenciales han conseguido una notable mejora en el tiempo de cómputo, consiguiendo una aceleración de hasta 800 veces respecto a su versión en Python puro. En el algoritmo de programación dinámica Bellman-Ford se obtuvo una evidente mejora de ejecución en un conjunto de distintos grafos; en uno de dichos grafos, formado con 2.500 vértices, se pasó de un tiempo de ejecución de alrededor de cincuenta minutos en su versión de Python a sólo un par de minutos de ejecución en Cython puro. En el algoritmo de Ford-Fulkerson, sin embargo, los tiempos obtenidos no fueron los esperados y, aunque sí se obtuvieron mejoras significativas en algunos de los casos, el uso natural de colas y la no homogeneidad de bucles e índices contribuyeron a un peor rendimiento.

Cython es, por tanto, un lenguaje fácil que oculta en su proceso de compilación la complejidad del lenguaje C y puede ser utilizado para obtener grandes resultados de aceleración de ejecución con respecto a un lenguaje exclusivo en Python.

Palabras clave: Cython, Pyrex, Python, aceleración, Bellman-Ford, Ford-Fulkerson, ecuaciones diferenciales, algoritmos

Abstract

Despite being dynamically typed, Python is a language with long runtimes, especially in algorithms that have many numbers. This is why specialised libraries such as h5py, scikit-image or Numpy, among others, are partially or highly developed in Cython.

This dissertation gives an introduction to the Cython language and studies the Bellman-Ford and Ford-Fulkerson algorithms, in which the runtimes are obtained for each of them in three different cases: with a pure Python version, with a Cython-compiled Python, and with a full Cython version. The differential equations are later analysed and the times are obtained with these three cases in the Lorenz equation system using Euler's Method.

The results obtained in solving differential equations have achieved a notable improvement in computing time, reporting an acceleration of up to 800 times compared to a pure Python version. In the Bellman-Ford dynamic programming algorithm, a clear improvement in execution was obtained in a set of different graphs. In one of these graphs, formed with 2,500 vertices, a runtime of around fifty minutes in Python was reduced to just a couple of minutes of runtime in pure Cython. However, the times obtained in the Ford-Fulkerson algorithm were not as expected and, although significant improvements were obtained in some of the cases, the natural use of queues and the non-homogeneity of loops and indices contributed to poorer performance.

Cython is therefore a simple language which hides the complexity of C in its compilation process and can be used to obtain major runtime acceleration results with respect to a language solely in Python.

Keywords: Cython, Pyrex, Python, acceleration, Bellman-Ford, Ford-Fulkerson, differential equations, algorithms

Índice general

Índice de figuras

Capítulo I

Introducción y objetivos

El objetivo principal de este capítulo es la presentación de mi Trabajo de Fin de Grado a través de un breve resumen de su contenido. Lo haremos mediante el desarrollo de dos puntos bien diferenciados: la motivación, donde se expone el ámbito de uso del programa Cython, su cuota de utilización actual y los objetivos, donde se describe la finalidad que se persigue con este trabajo y la estructuración intrínseca del documento, donde se explica en qué ha consistido cada capítulo.

1. Motivación

Físicos, matemáticos, biólogos, químicos, ingenieros... La ciencia avanza con la colaboración de personas formadas en distintos ámbitos. Todas ellas hacen uso de la alta capacidad computacional de los ordenadores que, con los algoritmos que nos ofrece el software, son imprescindibles para simular diferentes tipos de sistemas físicos y/o analizar la información obtenida. Esto los convierte en una herramienta esencial para el éxito.

Un lenguaje de programación intuitivo, sencillo y no "tipado"[1] puede ser el mejor candidato para su utilización en un ambiente multidisciplinar. Sin embargo, esta decisión conlleva la problemática de añadir un posible coste adicional: el tiempo de ejecución. Es por esto que Python, junto a algunas librerías como *NumPy*, *pandas*, *scikit-image* o *h5py*, es un candidato idóneo cuando hablamos de computación científica. Lo que se suele desconocer, es que todas estas librerías están escritas parcialmente (y por norma general en un alto porcentaje) en código Cython. Por tanto, el estudio de Cython nos podrá ayudar a conocer cómo funcionan estas librerías y, gracias a todas sus posibilidades, dispondremos de una herramienta potente y fiable.

El objetivo principal de este Trabajo de Fin de Grado es la introducción y aprendizaje de Cython a través de diferentes algoritmos e inferir, de forma empírica y a través de los resultados, en qué entornos conviene aplicar este útil lenguaje.

[1] En lenguajes dinámicos, como es el caso de Python o Perl, no es necesario decirle al compilador el tipo de dato que tienen asignado una variable y éstas pueden cambiar a lo largo del tiempo.

2. Organización de este documento

La organización se ha llevado a cabo en 4 bloques principales:

- Comenzamos con el **Capítulo II**, donde se presenta una breve introducción al lenguaje junto a algunas herramientas importantes de cara al estudio del código. Aquí se expone el origen de Cython, lo que nos llevará a hablar de sus inicios como Pyrex. Se comentarán las complejidades de desarrollar un módulo en C con la API de Python/C y analizaremos en qué nos podría beneficiar Cython. Se continúa con una introducción al lenguaje y, ya que Cython está ligado a C, explicaremos por qué conocer el lenguaje C es un requisito previo. A continuación se tratará su compilación junto a una explicación de algunas de las directivas más importantes del compilador para Cython. Se hará una pequeña exposición de lo que constituye un perfil (profile) y sus componentes, aplicando esta herramienta a un ejemplo real de un programa que utiliza el método de Montecarlo para calcular integrales. Familiarizados ya con esta herramienta veremos cómo generar el perfil y finalizaremos con la introducción de otra útil herramienta, timeit, para obtener tiempos de cómputo.

- El Trabajo de Fin de Grado continúa con el **Capítulo III**, donde se estudian dos importantes algoritmos en el ámbito de la computación, *Bellman-Ford* y *Ford-Fulkerson*. Se comienza desarrollando el uso de cada uno de ellos y las partes más importantes que usaremos de estos dos algoritmos. Se presentarán sus pseudocódigos, para convertirlos en su versión de Python puro, después haciendo uso de la herramienta profile se verá de forma empírica qué funciones son las más llamadas y, por tanto, las que más tiempo consumen. Después se finalizará con la conversión de este código en Python puro a su versión en Cython (con uso de memoria dinámica, librerías externas, etcétera). Se hará un cálculo de tiempos y una generación de gráficas donde se podrán analizar los diferentes tiempos obtenidos con la versión de Python, con su versión a través de Cython y, finalmente, con la versión Cython pura. Terminaremos el capítulo con unas conclusiones sobre los resultados.

- En el **Capítulo IV**, **ecuaciones diferenciales**, se hará una breve introducción a las ecuaciones diferenciales y su clasificación. Se detalla por qué constituyen una rica fuente de información si son representadas como un campo de pendientes y se estudiarán algunos métodos para su resolución numérica, entre ellos, el método de Euler, la versión mejorada de éste, y el método de Runge-Kutta. Se utilizará Cython para la resolución del plano x-z y se obtendrán los tiempos y la gráfica para el sistema de Lorenz. Por último, finalizaremos el capítulo con unas conclusiones.

- Para terminar, en el **Capítulo V**, se comentarán los resultados obtenidos a lo largo de este trabajo.

Capítulo II

Cython

> We follow two rules in the matter of optimization:
> Rule 1: Don't do it.
>
> *Michael A. Jackson*

El objetivo principal de este capítulo es presentar una introducción al lenguaje Cython, su compilación y algunas sencillas pero potentes herramientas como *profile* o *timeit* para obtener tiempos. También se verá la razón por la que *Gregory C. Ewing* desarrolló Pyrex (que más tarde se convirtió en Cython).

En el apéndice A y en el apéndice B se incluyen dos **entrevistas** con la participación y el consentimiento tanto de *Gregory C. Ewing*, desarrollador de Pyrex, como de *Stefan Behnel*, Core-developer de Cython, en exclusiva para este Trabajo de Fin de Grado.

1. Origen de Cython

Cython surgió de **Pyrex**, desarrollado por *Gregory C. Ewing*, un lenguaje para escribir módulos en Python de forma sencilla, el cual su autor lo define como *Python con tipos C*. El compilador Pyrex convierte el código Python a C utilizando la API C/Python pero teniendo en cuenta varias cosas, entre ellas el *recuento de referencias*, el uso de tipos de C (incluyendo punteros), que los valores Python y C se puedan entremezclar (si es posible) o la comprobación de errores junto el manejo de excepciones de Python.

Después Stefan Behnel junto a Robert Bradshaw y William Stein crearon Cython como un "fork" de Pyrex, pero que a día de hoy ya es una avanzada versión de éste. Cython siempre tuvo en mente continuar garantizando –al igual que hacía Pyrex– en la medida de lo posible que la sintaxis de Python fuera compatible y de esta manera el código fuera directo y limpio. Pero también incluye que el código escrito en Python pueda llamar de forma directa a librerías externas de C. Cython permite también especificar el tipo de las variables de Python, lo que producirá código C eficiente. A diferencia de Python, es necesario compilar los ficheros de Cython.

2. La programación de módulos en Python

La programación de módulos en C para Python es complicada, más aun si uno no tiene experiencia previa con la extensa *API Python/C*[14]. Se asume que se conoce de forma sólida el lenguaje C, no los punteros en sí, sino tablas virtuales y cómo uno puede hacer uso de ellas para no solo obtener poliformismo –dejando de ser algo exclusivo de los lenguajes orientados a objetos– sino que también el código resultante sea fácil de mantener y expandir. Después, como cualquier proyecto de gran envergadura, hay que seguir la correspondiente *guía de estilo* propuesta por Guido van Rossum y sin olvidar que antes de que pueda comenzar con el desarrollo de su módulo, será necesario escribir un poco de código (el llamado *boilerplate*).

Pero eso no es todo, deberá hacer frente y tener en cuenta también –pues el demonio está en los detalles– **el conteo de las referencias**. ¿Pero qué significa esto? Todo en Python es un objeto, incluido las funciones y enteros, es decir, estructuras en el heap que "heredan" la estructura base `PyVarObject`[15] junto a la compleja `struct _typeobject`, la cual se omite por su longitud.

Las dos estructuras más importantes que definen los modelos base de los objetos de CPython son las siguientes:

Código II.1: Todos los objetos en Python heredan de la estructura PyVarObject.

```
1  typedef struct _object {
2      _PyObject_HEAD_EXTRA
3      Py_ssize_t ob_refcnt;
4      struct _typeobject *ob_type;
5  } PyObject;
6
7  typedef struct {
8      PyObject ob_base;
9      Py_ssize_t ob_size; /* Number of items in
10                            variable part */
11 } PyVarObject;
```

La primera macro que encontramos, llamada `_PyObject_HEAD_EXTRA`, contenida en la estructura `PyObject`, sirve para propósitos de cara al debugging; cuando se expanda (si se encuentra definida la macro `Py_TRACE_REFS`) formará una lista enlazada doble incluyendo dos punteros: el siguiente y el previo de tipo `struct _object`. De esta forma, podremos tener una traza de todos los objetos del heap.

Cuando utilizamos *listas*, *tuplas* o *diccionarios* por mencionar los objetos más comunes, todos incluyen la macro `PyObject_VAR_HEAD` que será expandida por el preprocesador; es decir, un proceso previo y separado de la compilación que C proporciona para facilitar sustituciones como incluir el contenido de un fichero con el uso `#include` o remplazos de nombres, llamadas *macros* con el uso de `#define`.

Código II.2: Constante simbólica sustituida por el preprocesador de C.

```
1  #define PyObject_VAR_HEAD      PyVarObject ob_base;
```

Esto quiere decir que todos los objetos tendrán la información de su *tipo* definido por un puntero a la importante estructura `_typeobject` y la variable `ob_refcnt` que será la responsable de decirnos cuántos lugares hay que tienen una referencia a este objeto en particular. Cuando este conteo llega a cero, el objeto se libera. Este recuento ha de hacerse de forma explícita usando dos macros llamadas `Py_INCREF()` para incrementar en uno y `Py_DECREF()` para decrementar en uno. Cuando llega a cero, la estructura `_typeobject` contiene un puntero a una función llamada `tp_dealloc`, que será la responsable de liberar de forma correcta ese tipo de objeto en particular y si por ejemplo, fuera una lista u otro tipo de objetos compuestos (como diccionarios), las referencias que pueda tener éste.

Hay situaciones en las que tampoco es necesario incrementar este conteo (por ejemplo en variables locales que apunten a un objeto), pues cuando salgamos del alcance/ámbito de esa variable habrá que decrementar de nuevo y al final el conteo final no habría cambiado. Hay una serie de buenas prácticas para familiarizarse con el uso correcto del conteo, entre ellas cuando los objetos se pasan a funciones C en un módulo llamado desde Python; ahí deberemos mantener una referencia a cada argumento durante la duración de la llamada. O es posible olvidar incrementar en uno cuando extraemos un objeto de una lista; podría otra operación eliminar este objeto, con el correspondiente decremento y finalmente –si llegara a cero– con su liberación.

Hay buenas razones por que uno se podría adentrar en el desarrollo de módulos en C para Python: primero, porque la funcionalidad que se quiere utilizar en Python no existe o podríamos querer añadir un nuevo tipo de objeto para Python; segundo, porque los módulos serían mucho más rápidos que si fueran realizados en Python (un $10\,\%$ a $30\,\%$ más); y por último y no por ello menos importante, por aprender temas nuevos que lleven un desafío implícito con el añadido de poder participar en la comunidad del *software libre* de Python.

Ahora bien, si de lo que se trata es de obtener la máxima aceleración, no sería buena idea desarrollar nuestro propio módulo, pues la falta de experiencia en la API de C de Python conllevaría un largo proceso de aprendizaje y testing. Otra posibilidad si uno se siente cómodo en C, sería programarlo en puro C, superaría en velocidad al módulo, pero sería un trabajo en vano al no poder llamar al algoritmo resultante desde Python. Es aquí donde aparece Cython, para darnos acceso al mundo de Python y C; con la sintaxis de Python y con la posibilidad de añadir tipos a nuestras variables, habremos superado con creces a nuestro novel módulo en un corto plazo de tiempo. Otra de las ventajas de Cython es su licencia de software libre, por lo que su desarrollo está en constante evolución[1]; la comunidad corrige bugs, optimiza o implementa nuevas funcionalidades, lo que garantizará una mayor aceleración y estabilidad respecto a un módulo realizado a mano sin experiencia previa.

[1]Esta evolución es natural desde el mismo instante que un proyecto conocido se libera; sin embargo, a mi juicio, el software libre es más que este pragmático punto.

3. Un Python extendido

Cython es un lenguaje que se encuentra entre dos mundos. Tiene la sencillez de un lenguaje de alto nivel como Python (pues es un superconjunto de éste), con el extra de producir código muy optimizado de bajo nivel como es C.

Para alguien que este familiarizado con C, la sintaxis de Cython es bastante sencilla, basta con añadir la palabra reservada `cdef` cuando se quiera definir variables en C. Por ejemplo:

```
1  cdef int x, float y
```

Tambien hay disponibles tipos básicos como **`char`**, **`short`**, **`long`**, **`long long`** y sus versiones sin signo `usigned` **`int`**, y, cómo no podia ser de otra forma, punteros.

En funciones, lo importante es tener en mente que hay dos tipos de definiciones de funciones en Cython: las normales utilizando **`def`** (como se hace en Python), que toman objetos Python como parametros y devuelven objetos Python; y las funciones C usando `cdef`, que toman tanto objetos Python como valores en C y pueden devolver tanto uno como otro. Esto se traduce en que dentro de un módulo de Cython, los dos tipos de definiciones se pueden llamar entre ellas libremente. Sin embargo, cuando queramos hacer uso de alguna de estas funciones con el intérprete de Python, solo podrán ser vistas las funciones Python (definidas **`def`**). Existe otra palabra reservada en Cython que es aplicable en las definiciones de funciones: `cpdef`, vista como un hibrido, y que las funciones definidas de esta forma, se podrán ver desde cualquier lado, pero añadirá una sobrecarga.

Referente a punteros, al igual que en C, el operador unario * llamado de indirección (o desreferencia) va con su variable. Por ejemplo, en el siguiente código, la variable `mi_ptr_entero` es la única declarada como puntero a **`int`**, la otra, `mi_entero` es un entero. El operador inverso de indirección &, que da la dirección de una variable, es también legal en Cython. Para acceder al valor de una variable desde su puntero o para modificarla, no podemos utilizar de nuevo el operador de indirección sino utilizar el indice cero.

```
1  cdef int *mi_ptr_entero, mi_entero
2  mi_ptr_entero = &mi_entero
3
4  # para modificar/acceder al valor
5  # debemos hacerlo con índices
6  mi_ptr_entero[0] = 1234
7  print(mi_ptr_entero[0])
```

Se incluye también un tipo especial para valores boleanos:

```
1  cdef bint is_updated
```

Veamos un sencillo pero esclarecedor ejemplo de lo que podríamos conseguir con lo visto en estos momentos con Cython. Tengamos un bucle normal en Python como el siguiente:

```
1  for i in range(10):
2      pass
```

A primera vista parece inofensivo, pero si lo miramos con un poco más de detalle y ahondamos en las entrañas del intérprete CPython (que no Cython), podremos observar cómo está implementada la función **range** detrás del dinamismo de Python. Se muestra de forma resumida a continuación:

```
1  static PyObject *
2  range_new(PyTypeObject *type, PyObject *args, PyObject *kw)
3  {
4
5      # comprobar tipos, conversiones, etcétera.
6  }
```

Se ve cómo recibe un objeto `PyTypeObject`, que esto era como se vio antes, el tipo de dato `struct _typeobject`. Aparte también hay conversiones de tipos, chequea los tipos, hay llamadas a punteros a funciones, etcétera.

Ahora se muestra el pequeño cambio introducido en el bucle orginal de Python, añadiendo un tipo estático entero con `cdef`

```
1  cdef int i
2  for i in range(10):
3      pass
```

Y si este código lo compilamos –se ve en el siguiente punto del capítulo– para ver qué código C generará Cython, se tiene:

```
1  for (__pyx_t_1 = 0; __pyx_t_1 < 10; __pyx_t_1+=1) {
2      __pyx_v_8ejemplo1_i = __pyx_t_1;
3  }
```

En el fichero *C* generado se puede encontrar[2] la variable `__pyx_v_8ejemplo1_i` (privada) y `__pyx_t_1` (ámbito local) han sido declaradas como **`static int`** e **`int`** respectivamente. Ahora son enteros, no hay tipos ni comprobaciones por parte de CPython. Es decir, en un código de Python, cuando hay por ejemplo dos enteros que se suman $a+b$ Python no sabe que son enteros, o que deben ser tratados como enteros, por tanto, deberá preguntar por su *tipo* (¿es un entero? ¿es una lista? ¿o quizás un diccionario?...), buscar una función del tipo `__add__` para saber cómo actuar, habiendo pasado la información como argumentos, luego extrayendo la información (enteros) de las estructuras con *MACROS de la API* para convertir Python a C. Si todo va bien, convertirá el resultado en otro objeto Python. En todo este proceso, ha habido toda una larga serie de llamadas a funciones, punteros, castings y máquinas virtuales.

En Cython, no habrá una máquina virtual (*VM*) de Python leyendo *bytecode* y ya conocerá su tipo al haberlo tipado de antemano y, por tanto, esto producirá una instrucción máquina directa para el procesador. El código C producido será eficiente también.

[2]La salida se omite en este documento debido a la cantidad de líneas.

En Cython está permitido declarar estructuras, uniones y enumeraciones:

```
cdef struct comando:
    char *nombre
    int (*fnc_comando)(const char **arg)

cdef union u_tag
    int ival
    float fval
    char *sval

cdef enum Tipo:
    TIPO_FOO = 0,
    TIPO_BAR,
    NUM_MAXIMO_TIPOS
```

Cython soporta la programación orientada a objetos. Si tenemos una clase en Python podremos compilarla con Cython; sin embargo, el código C resultante será una clase regular con un uso grande de llamadas de la API C de Python, la mayor parte de ellas referentes a conversiones (pues una variable Python puede ser cualquier cosa en cualquier momento).

Con añadir el tipado (tanto delante de la palabra reservada `python` como en las variables), le decimos a Cython que trate a esta clase de forma tipada en vez de una regular eliminando gran parte de la sobrecarga que añadía Python de forma dinámica.

```
cdef struct Punto:
    float x, y

cdef class Esfera:
    cdef Punto pnt
    cdef float radio
        ...
```

Conviene recordar que hay diferencias respecto al lenguaje C. Por ejemplo, los *castings*: en C se usa (y), mientras que en Cython se usa < y >. Un ejemplo:

```
cdef char *ptr_mi_char
cdef int *ptr_mi_entero
ptr_mi_char = <char *>ptr_mi_entero
```

Uno de los principales usos de Cython es la utilización de librerías de C externas, esto permite delegar la mayor carga del cómputo del algoritmo en librerías de C y se obtiene, de esta forma, el resultado final en la función de Python original. Esta técnica, consistente en el empleo de funciones ajenas para incrementar la celeridad, se denomina **wrapper** y se fundamenta en el uso de funciones definidas en `def` y que, a través de las correspondientes funciones en Cython, será la responsable de la aceleración del programa (definidas con `cdef`).

Por ejemplo si tuvieramos una función definida en C procederíamos a dotar al fichero de cabecera con `cdef extern from`. Así la plantilla nos quedaría de la siguiente forma:

```
cdef extern from "mi_fichero_cabecera.h":
    void mi_funcion_1(int a, int b)
    float mi_funcion_2(mi_struct var1)
```

Después habría que definir una función de Python puro que será la responsable de la llamada a la otra función. Esta función hará de wrapper.

```
1  def mi_wrapper():
2      mi_funcion_1(3, 4)
```

Toda la definición de funciones es similar a la de C:

```
1  def mi_wrapper(unsigned int ui, char *str):
2      ...
3
4  cdef float bar(unsigned int ui, char *str):
5      ...
```

Se permiten las funciones `inline`, las cuales son recomendadas para pequeñas operaciones dentro de bucles.

```
1  cdef inline Vector sumar_vector(Vector a, Vector b):
2      ...
```

En Cython existen 2 tipos de ficheros principales: los que tienen extensión **.py** y los **.pyx**. Los dos contienen las implementaciones de funciones, las clases, etc, y que Cython podrá compilar sin problemas; sin embargo si queremos hacer uso de la sintaxis de Cython deberemos usar de forma obligatoria la extensión **.pyx**. Cada vez que Cython compila un fichero .pyx comprueba si existe su correspondiente fichero con extensión **.pxd** y lo procesa primero. Sería el equivalente a los ficheros de cabecera **.h** de C conteniendo las declaraciones.

4. Compilación

Los ficheros de Cython se componen por el nombre del módulo seguido por la extensión *.pyx*. El compilador de Cython convierte todo el código contenido en el fichero.pyx a otro fichero en *C*. Después este código C se convierte a una librería dinámica (*.so*) mediante un compilador de C (por ejemplo *gcc*).

Es fácil ver la sálida en C que produce Cython: bastaría con crear un fichero *test.pyx* añadir algo de código en Python/Cython y compilarlo de forma manual. Se ve a continuación:

```
1  $ cat test.pyx
2  cdef int var1 = 4
3  var2 = 5
4  lista = [44, 45]
5  $ cython -a test.pyx
```

Esto generará dos ficheros llamados **test.c** y **test.html**. El primero, será un fichero C que contendrá unas 2.500 líneas, en el que incluirá comprobaciones de errores/overflows, conversión de tipos entre C y Python y en general estructuras como diccionarios

```
+1: cdef int var1 = 4
+2: var2 = 5
+3: lista = [44, 45]
```

(a) Diferentes grados de uso de la API de Python.

```
+1: cdef int var1 = 4
    __pyx_v_4test_var1 = 4;
+2: var2 = 5
    if (PyDict_SetItem(__pyx_d, __pyx_n_s_var2, __pyx_int_5) < 0) __PYX_ERR(0, 2, __pyx_L1_error)
+3: lista = [44, 45]
    __pyx_t_1 = PyList_New(2); if (unlikely(!__pyx_t_1)) __PYX_ERR(0, 3, __pyx_L1_error)
    __Pyx_GOTREF(__pyx_t_1);
    __Pyx_INCREF(__pyx_int_44);
    __Pyx_GIVEREF(__pyx_int_44);
    PyList_SET_ITEM(__pyx_t_1, 0, __pyx_int_44);
    __Pyx_INCREF(__pyx_int_45);
    __Pyx_GIVEREF(__pyx_int_45);
    PyList_SET_ITEM(__pyx_t_1, 1, __pyx_int_45);
    if (PyDict_SetItem(__pyx_d, __pyx_n_s_lista, __pyx_t_1) < 0) __PYX_ERR(0, 3, __pyx_L1_error)
    __Pyx_DECREF(__pyx_t_1); __pyx_t_1 = 0;
```

(b) Desplegando, veremos qué ha hecho el compilado de Cython a C.

*Figura II.1: Contenido del fichero **test.html**.*

para mantener el dinamismo[3] de las variables y toda la declaración y uso de primitivas necesarias para la inicialización de módulos C. Parte de este código, como las comprobaciones de errores, se pueden eliminar utilizando directivas del compilador, como se verá más adelante.

El segundo fichero, será un reporte en formato html como se ve en la Figura II.1 que facilita ver el nivel de inclusión de la api de C para Python. Es decir, cuando Cython compila el código Python, genera toda una serie de llamadas a la API de CPython, lo que producirá lineas amarillas, repercutiendo en la velocidad de ejecución; por otro lado veremos lineas blancas si ha compilado con puro C, sin llamadas a la API de Python, lo cual mejorará la velocidad del código. Si expandimos la información veremos el uso de Cython de la API de Python. En la figura, `var1` es una variable pura estática de tipo entero; para `var2` se emplea un diccionario de la API de Python, pues `var2` podría más adelante cambiar de tipo; tambien se ha realizado una conversion entre tipos (de entero con valor 5 a puntero `PyObject`).

Para la variable `lista` se reserva espacio para los dos elementos, actualizando el conteo de referencias. Se hace uso de la función `PyList_New` que necesita por argumento el número de elementos para reservar su memoria (en este caso dos elementos) y mantiene en todo momento el recuento de referencias, habiendo de antemano realizado la conversión entre tipos C y Python.

Cython nos facilita todo el proceso de compilado hasta obtener una librería dinámica (cómo no, de extensión **.so**) que podremos importar directamente en el intérprete de Python (llamado CPython). Este proceso de compilado se hace con el comando *cytho-*

[3]Por ejemplo una variable puede comenzar siendo un entero y despues ser una lista.

nize:

```
$ cythonize -a -i test.pyx
```

Este comando nos creará los mismos ficheros que se generaron con el comando `cython` , esto es, el fichero C y su salida html (para facilitar la lectura del código). Pero también realizará la compilación del fichero C (usando por ejemplo gcc) y creando la correspondiente librería dinámica. Una vez hecho esto, la librería estará disponible para importarla en Python como si de cualquier otro módulo se tratara:

```
import test
```

5. Directivas del compilador

Las directivas del compilador son instrucciones que afectan al comportamiento del código de Cython dependiendo si están activadas o no. Vamos a comentar en detalle dos.

Un primer ejemplo es cuando Cython, al compilar el código a C, añade toda una serie de comprobaciones para verificar que no accedemos a posiciones de memoria de un array que están fuera de los limites. En este caso, si estamos seguros de que hemos depurado correctamente el código, no es necesario que el procesador pierda tiempo realizando unas operaciones que nunca resultarán en excepciones de Python y podremos desactivar las comprobaciones.

Esta en particular podría desactivarse para todo el código con este comentario en la parte superior del fichero:

```
# cython: boundscheck=False
```

o solo aplicarla para una función concreta mediante un *decorator*:

```
#!python
cimport cython

@cython.boundscheck(False)
def foo():
    pass
```

Como segundo ejemplo, algunas veces en Python es natural acceder a los elementos de los arrays utilizando índices negativos. Cython también soporta esto. Pero si el código no presenta este uso de índices, podremos desactivar la correspondiente directiva para que no haya código C extra manejando esta lógica. Esta directiva es manejada por:

```
@cython.wraparound(True/False)
```

6. La herramienta profile

Uno de los errores más comunes en el que podemos caer cuando uno se inicia en Cython es sobresaturar de forma obsesiva todo el código con tipos estáticos. Esto podría conducir a que se pueda romper la ganancia de velocidad que Cython puede aportar, pues éste internamente optimiza de forma muy rigurosa el código, y si uno hace mal uso de los tipos e índices, es posible obtener más desventajas que ventajas. Más grave sería si a todo esto añadimos otras estrategias para acelerar el código sin llegarse a preguntar si en última instancia supondrá alguna diferencia en cuanto a reducción de tiempos. Por eso, una primera aproximación, por ejemplo, para códigos no muy extensos se podría hacer con una primera batería de pruebas, aplicar los conocimientos aprendidos y ver de forma empírica los tiempos. Esto cambia cuando tenemos proyectos complejos y extensos donde la inclusión de tipos en variables y funciones no solo será una perdida de tiempo –en cuanto a conseguir mejoras de rendimiento respecto al código original y lo que supone la modificación de todos los ficheros– sino que también en el código resultante se perdería parte de la liviana sintaxis *pythonica* que Python nos aporta. Sin embargo, hacer una prematura optimización a una función elegida por nuestro equivocado juicio y que no sea la responsable del rendimiento final de nuestro código nos llevará a no aprovechar toda la potencia de Cython.

La clave es, por tanto, realizar un estudio de nuestro código con un *perfil* y comprobar con pensamiento crítico qué funciones serán las prioritarias para su modificación. Y ahí es donde entra el fichero `profile.py`, un pequeño script que nos ofrecerá todo un conjunto de estadísticas sobre con qué frecuencia y durante cuánto tiempo se ejecutan varias funciones del programa. Esta información será crucial a lo largo de este Trabajo de Fin de Grado.

Pero obtener esta información tiene un coste, y es que la carga de perfiles añade una sobrecarga de tiempo a las funciones. Aclarar que este tiempo no se añade dentro del tiempo gastado dentro de la función sino que se incluye en la propia llamada. Esta carga puede ser importante si tenemos funciones pequeñas que se llaman de forma frecuente.

Veamos un ejemplo de la información que contiene un perfil real. Para ello se realizó un pequeño programa mostrado en el Código II.3, donde se hace uso de un método de Monte Carlo para resolver el cálculo del volumen aproximado de una esfera centrada en el origen y de radio 1. La idea de Monte Carlo gira entorno a la probabilidad de que un punto, generado al azar, se encuentre dentro de la esfera o fuera. La esfera se encuentra contenida en un cubo del mismo radio que la esfera. Por tanto, si generamos N puntos dentro del cubo se espera que $N * V_{esfera}/V_{cubo}$ puntos, estén dentro de la esfera. Por tanto, el volumen de la esfera vendrá dado por la siguiente fórmula:

$$V_{esfera} = V_{cubo} * \frac{Puntos_{esfera}}{Puntos_{totales}} \tag{1}$$

Para los $Puntos_{totales}$ de la ecuación (1) se generan $10.000.000$ de puntos dentro del cubo de forma aleatoria para la aproximación del volumen. Para saber si los puntos están dentro o fuera de la esfera se hace uso de la función `esta_dentro_esfera` que devuelve un valor `True` si se encuentra dentro o `False` en caso contrario.

Código II.3: Cálculo del volumen de una esfera en Python con el método de Monte Carlo.

```
1  import random
2
3  def esta_dentro_esfera(x, y, z):
4      d = x*x+y*y+z*z
5      return d < 1.0
6
7  def calculo_volumen():
8      dominio_x = [-1.0, 1.0]
9      dominio_y = [-1.0, 1.0]
10     dominio_z = [-1.0, 1.0]
11
12     volumen = (dominio_x[1]-dominio_x[0])*\
13               (dominio_y[1]-dominio_y[0])*\
14               (dominio_y[1]-dominio_y[0])
15
16     muestras = 10000000
17     puntos_dentro = 0
18     for i in range(muestras):
19         x = dominio_x[0] + (dominio_x[1]-dominio_x[0])*random.uniform(0,1)
20         y = dominio_y[0] + (dominio_y[1]-dominio_y[0])*random.uniform(0,1)
21         z = dominio_z[0] + (dominio_z[1]-dominio_z[0])*random.uniform(0,1)
22
23         if (esta_dentro_esfera(x, y, z)):
24             puntos_dentro = puntos_dentro+1
25
26     fraccion = puntos_dentro/muestras
27     integral = volumen*fraccion
28     return integral
29
30 def main():
31     v = calculo_volumen()
32     print("Volumen aproximado de la esfera (Radio=1): ", v)
```

En la Figura II.2 se puede ver el perfil del Código II.3. La primera información que se observa en este perfil es el número total de llamadas, en este caso, dieron lugar a $70.000.006$ de llamadas en un tiempo de $11{,}760$ segundos. La primera columna, de nombre *ncalls*, contiene el número de llamadas en total para cada función individual. Las columnas más interesantes son *tottime* y *cumtime*. La primera es el tiempo gastado dentro de la función sin contar llamadas a otras funciones dentro de ésta. La segunda es igual que *tottime* pero incluyendo el tiempo de las otras llamadas. Es de esperar que el número de llamadas a la función `uniform` del módulo `random` sea tres veces más.

6.1. Cómo generar un profile

Ahora que estamos familiarizados con la salida por pantalla de un profile, veamos las partes que contiene dicho fichero. El código del script que calcula un profile se puede encontrar en el apéndice C.

Figura II.2: Profile obtenido del programa para el cálculo del volumen de una esfera.

```
Volumen aproximado de la esfera (Radio=1):  4.1882272

         70000006 function calls in 11.760 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    5.716    5.716   11.760   11.760 montecarlo.py:7(calculo_volumen)
 30000000    4.021    0.000    5.046    0.000 random.py:371(uniform)
 30000000    1.025    0.000    1.025    0.000 {method 'random' of '_random.Random' objects}
 10000000    0.998    0.000    0.998    0.000 montecarlo.py:3(esta_dentro_esfera)
        1    0.000    0.000   11.760   11.760 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000   11.760   11.760 montecarlo.py:30(main)
        1    0.000    0.000   11.760   11.760 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Lo primero será la importación de dos módulos que son `pstats` y `cProfile` [4]. Luego se deberá añadir el módulo del código que queremos analizar, en este caso `montecarlo`.

El constructor de la clase `pstats.Stats` será el encargado de leer de un fichero (o una lista de ficheros) generado por la instancia de la clase cProfile.

El constructor de `Stats` es el siguiente:

```
1  class pstats.Stats(*filenames or profile, stream=sys.stdout)
```

El método `runctx` de la clase `cProfile` hará una llamada a la función de Python **`exec`** que ejecutará el modulo importado.

7. La herramienta timeit

Existe otro pequeño módulo muy útil para aislar pequeños segmentos de código Python/Cython y obtener precisos tiempos. Este módulo se ha usado para la medición de tiempos en el capítulo de ecuaciones diferenciales. Este módulo es `timeit` y se puede usar tanto desde el intérprete Python como con el intérprete interactivo de Python **IPython**.

Veamos cómo utilizar este comando junto al intérprete. Para ello supondremos la situación de que tenemos dos ficheros, uno de ellos con nombre **python_fib.py** con la función de Fibonacci en Python puro y en el otro fichero con nombre **cython_fib.pyx** la misma función en Cython.

[4] También se podría en su sustitución utilizar el módulo `profile`; sin embargo éste último es Python puro y añadirá una importante *overhead* (sobrecarga) al perfil.

Código II.4: Función de Fibonacci en Python puro.

```
1  def fib(n):
2      a = 0
3      b = 1
4      for i in range(n):
5          a, b = a+b, a
6      return a
```

Código II.5: Función de Fibonacci en Cython.

```
1  def fib(int n):
2      cdef int a = 0
3      cdef int b = 1
4      cdef int i
5      for i in range(n):
6          a, b = a+b, a
7      return a
```

La forma para utilizar *timeit* desde el intérprete se muestra en el Código II.6 (versión Python puro) y en el Código II.7 (versión Cython). En el siguiente capítulo se estudiará la forma para proceder a la compilación de código en Cython.

Código II.6: Obtención de tiempos con el módulo timeit en el cálculo de la función en Python puro de Fibonacci.

```
1  $python3.6 -m timeit --unit sec -s "import python_fib" "python_fib.fib(100)"
2  100000 loops, best of 3: 3.03e-06 sec per loop
```

Código II.7: Obtención de tiempos con el módulo timeit en el cálculo de la función en Cython de Fibonacci.

```
1  $python3.6 -m timeit --unit sec -s "import cython_fib" "cython_fib.fib(100)"
2  10000000 loops, best of 3: 7.23e-08 sec per loop
```

En el siguiente capítulo se aplicará lo visto en estas secciones para utilizar Cython en la resolución de problemas de programación dinámica.

Capítulo III

Aplicación de Cython en programación dinámica

> If you can solve it, it is an exercise; otherwise it's a research problem.
>
> *Richad Bellman*

El objetivo principal de este capítulo es presentar en detalle dos algoritmos de programación dinámica, desarrollar su código en Python y después hacer su versión en Cython, una de ellas código Python compilado directamente a través de Cython y otra añadiendo tipos y aplicando algunas directivas de compilador. Luego, se generarán algunos grafos de forma aleatoria con distintos tamaños y con esta información se recogerán sus tiempos y se analizarán los mismos.

El primer algoritmo que se estudiará será **Bellman-Ford**[1], incluido en la familia de problemas del camino más corto; el otro es **Ford-Fulkerson**, catalogado dentro de los algoritmos de flujo. La naturaleza de ambos, al representar grafos, obligará a organizar la información en listas o matrices de adyacencia. En este Trabajo de Fin de Grado se ha elegido matrices para la implementación de ambos algoritmos. Una vez conseguido esto, cada uno de estos algoritmos deberá trabajar de forma intensa y muy diferente con estas matrices para obtener su respectiva solución.

Por ejemplo, Bellman-Ford, deberá aplicar una técnica de relajamiento –se verá en profundidad más tarde– a cada par de vértices, lo que implicará iterar sobre todos los vértices. Ford-Fulkerson, por otro lado, hará uso y de forma repetida del algoritmo de *búsqueda en anchura* para encontrar los cáminos en un *grafo residual*.

1. Algoritmo de Bellman-Ford

En el algoritmo de Bellman-Ford, al comprobarse todos los pares de vértices de forma intensiva por cada fase –lo que conlleva varios bucles anidados– se tiene un excelente candidato para ver en detalle con Cython.

Este algoritmo resuelve el mismo problema que el algoritmo de Dijkstra, esto es,

[1] También conocido como Bellman–Ford–Moore, al haberse publicado por este último dos años después.

devolver tanto la tabla de los vértices precedentes de los caminos más cortos, como la tabla de distancias. La diferencia es que Bellman-Ford permite grafos con costes negativos, siempre que no haya **ciclos** negativos. En el caso de que los hubiera, no podríamos obtener una solución pero sí sabríamos que hemos detectado uno.

Uno de los principios clave en que se basa la familia de soluciones del camino más corto, es la técnica de la relajación de caminos que se verá más adelante. Estos tipos de problemas se representan por grafos $G = (V, E)$, dirigidos y con pesos, donde éstos vienen dados por una función de peso $w : E \to \mathbb{R}$. Ahora se definirá la función de *coste de un camino* $w(p)$:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \tag{1}$$

donde $p = \langle v_0, v_1, ..., v_k \rangle$ es un cámino. El resultado de esta función será la suma de todos los pesos de las aristas entre los vértices v_0 a v_k.

Ahora, la distancia mínima de un vértice u a v vendrá dado por la función $\delta(u, v)$ y se define como:

$$\delta(u, v) = \begin{cases} min\{w(p) : u \xrightarrow{\text{p}} v\} & \text{Si hay un camino de } u \text{ a } v. \\ \infty & \text{En caso contrario.} \end{cases} \tag{2}$$

Los pesos negativos en un grafo son naturales y pueden con ello modelar otro tipo de situaciones. Tanto con pesos negativos o sin ellos la ecuación (2) estará bien definida. Sin embargo cuando un grafo $G = (V, E)$ contiene un bucle negativo y u, v son vértices en él, no habrá ruta más corta entre u y v, pues siempre podremos encontrar otro camino con un peso inferior siguiendo el camino original y después atravesar varias veces por el ciclo, terminando en un bucle con un coste $-\infty$. Si ocurre esta situación, en la que hay un ciclo negativo en algún camino de u a v, entonces $\delta(u, v) = -\infty$.

1.1. El método de relajación

Como ya se ha dicho, el corazón de los algoritmos de problemas de caminos más cortos (como por ejemplo Dijkstra y Bellman-Ford) se basa en la técnica de relajación; la diferencia entre ellos es la cantidad de veces que la utilizan para cada vértice y en el orden en el que lo hacen. Su principio es minimizar las distancias de caminos más cortos de un vértice origen s a cualquier vértice $v \in V$. A estos valores se les llama *distancias de camino más corto*. El algoritmo para la operación de relajación comienza con el Código III.1, donde se empezará iterando sobre todos los vértices del grafo, inicializando las distancias a ∞ a excepción del vértice s (origen) que estará a 0. También se asignará a todos los vértices su predecesor el valor `NULL`.

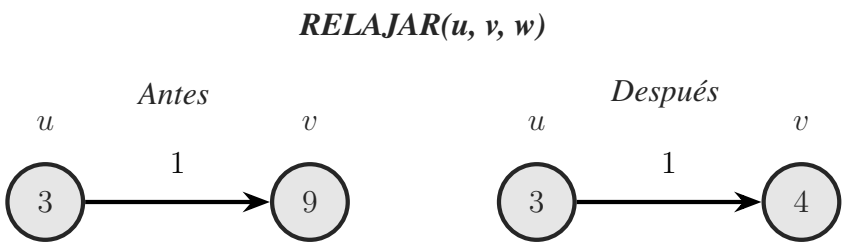


Figura III.1: Relajar un vértice se realiza en coste $\mathcal{O}(1)$. Si la condición no se hubiera dado, por ejemplo si el coste del vértice v hubiera sido 2 en vez de 9 la condición habría sido falsa al ser $2 > 3 + 1$.

Código III.1: Pseudocódigo para la operación INICIALIZAR.

```
1  INICIALIZAR(Grafo, source)
2      Para cada vértice v ∈ Grafo.V inicializamos con:
3          v.d  = ∞
4          v.p = NULL
5      source.d = 0
```

El coste de la operación *INICIALIZAR* visto en el Código III.1, será de $\Theta(|V|)$.

Una vez establecidos los valores iniciales, se hace uso de la operación *RELAJAR* que comprueba, como se aprecia en el Código III.2, si la distancia actual de un vértice v es mayor que la suma de la distancia al vértice precedente más el peso $w(u, v)$, es decir, es preguntarse si podemos encontrar un camino más corto que otro ya conocido.

Código III.2: Pseudocódigo para la operación RELAJAR.

```
1  RELAJAR(u, v, w)
2      si v.d > u.d + w(u,v)
3          v.d  = u.d + w(u,v)
4          v.p  = u
```

En la Figura III.1 se observa un sencillo ejemplo de una actualización de distancias para dos vértices cualesquiera u y v antes de que se aplique el método RELAJAR y después. Bellman-Ford hará uso de esta operación en cada arista $|V| - 1$ veces.

1.2. Pseudocódigo y código

En esta parte se verá algunas partes críticas del ćodigo necesarias para la ejecución del algoritmo Bellman-Ford. Se explicará en detalle primero el pseudocódigo del algoritmo Bellman-Ford completo que estará constituido por todas las pequeñas partes que se han visto anteriormente (inicialización, relajamiento) junto a la detección de ciclos negativos y después se procederá a ver las partes individuales (y más importantes) del

código en Python puro. Para finalizar, se clasificarán esas partes en Python puro y se realizará la conversión en Cython explicando los cambios realizados.

El código completo de la versión pura en Python para el algoritmo de Bellman-Ford se puede encontrar en el apéndice E y su versión en Cython en el apéndice F.

1.2.1. Pseudocódigo

En el pseudocódigo del Código III.3 se observa que no se realizan las $|V|$ iteraciones completas. La razón de esto es porque cualquier camino con $|V|$ o más aristas debe tener un ciclo. Las líneas 4-6 relajan cada arista del grafo una vez. Las líneas 11-14 comprueban que no haya un ciclo negativo. El algoritmo se ejecuta en tiempo proporcional $\mathcal{O}(VE)$, pues se realizan V pasadas a través de todas las aristas E.

Código III.3: Pseudocódigo para el algoritmo Bellman-Ford.

```
1  BELLMAN-FORD(G, s, w)
2      INICIALIZAR(G, s)
3
4      Para cada i = 1 hasta |V| - 1
5          Para cada arista (u,v) ∈ G.E
6              RELAJAR(u, v, w)
7
8
9      # si se detectan ciclos negativos
10     # se devuelve FALSE
11     Para cada arista (u,v) ∈ G.E
12         Si v.d > u.d + w(u,v)
13             return FALSE
14     return TRUE
```

1.2.2. Código en Python puro

Una de las grandes ventajas de Python es que la sintaxis se asemeja al pseudocódigo, lo que facilitará la lectura del código de las diferentes partes. Se procede a analizar sólo algunas partes del código que más tarde se transformarán a Cython. El código completo del algoritmo de Bellman-Ford en Python puro se puede encontrar en el apéndice E.

Se partió de un diseño orientado a objetos en el cual hay una clase `MatrizAdyacencia`, que se puede instanciar recibiendo una matriz `numpy` a medida (con sus pesos y sus adyacencias) a través del `constructor` de la clase, o bien, se puede generar una de forma aleatoria llamando al método `aleatoria` y estableciendo una serie de atributos: el número de vértices, un factor de densidad `factor` $\in [0, 1]$, donde 1 daría un *grafo completo*, y un intervalo de pesos. Esta clase tiene un método privado llamado `_convert_to_dic` que convierte esa matriz de adyacencias a un diccionario para acceder en tiempo constante a los vértices adyacentes de cada vértice. Cada una de estas claves en el diccionario (vértices) apunta a una lista donde cada elemento tiene una tupla (inmutable) conteniendo el vértice adyacente junto al peso a éste. La clase `MatrizAdyacencia` al ser la responsable

de la representación de un grafo, será una instancia de la clase `BellmanFord` que recibirá a través del `constructor` o por medio del método `set_obj_matriz`, teniendo así no sólo toda la información necesaria para su ejecución sino, también, separando la lógica de los datos.

Para la clase `BellmanFord` hay tres principales métodos. Uno de ellos es el responsable de la inicialización del algoritmo que se encuentra en el Código III.4. Se puede observar cómo se inicializa la distancia del vértice *origen* (s) a 0 y el resto de vértices a ∞. En la misma iteración se establecen los previos a `NULL` con la palabra reservada en Python `None`.

Código III.4: Python puro para la operación INICIALIZAR.

```
1  def _iniciar_relax(self):
2      num_vertices = self.m.get_num_vertices()
3      for i in range(num_vertices):
4          self.d.append(np.inf)
5          self.p.append(None)
6      self.d[self.s] = 0
```

El otro método es el mostrado en el Código III.5, donde se aplica la operación de *relajamiento* mostrada en la Figura III.1. Se ve como el método recibe dos vértices y un peso. Accedemos a las distancias de cada uno de los vértices y las guardamos en variables locales para una mejor lectura. Comprobamos si podemos relajar el coste de s a v, actualizando el previo a través del cual podemos rebajar este coste y la distancia nueva obtenida. Si no se pudiese aplicar esta operación, se quedaría todo como estaba.

Código III.5: Python puro para la operación RELAJAR.

```
1  def _relax(self, u, v, w):
2      u_d = self.d[u]
3      u_v = self.d[v]
4      if u_v > u_d + w:
5          self.d[v] = u_d + w
6          self.p[v] = u
```

Por último, el código mostrado por el Código III.6 es el cuerpo principal del algoritmo BellmanFord. Se obtiene el número total de vértices del grafo a calcular. Después se hace una llamada al método privado `_iniciar_relax` que ya ha sido explicado previamente en el Código III.4. Luego, en las líneas 6 a 9 se itera $|V| - 1$ veces, esto es, se recorre el número total de vértices menos uno. Para cada uno de estos vértices se obtienen sus adyacentes consiguiendo la distancia y el coste de cada uno de ellos con el fin de poder aplicar el método de relajamiento a cada uno posteriormente (de ahí el doble bucle anidado). Finalmente se realiza la detección (líneas 11-14) de ciclos negativos y en el caso de encontrar uno, el algoritmo se detiene.

Código III.6: Python puro para la operación EJECUTAR.

```
1  def ejecutar(self):
2      num_vertices = self.m.get_num_vertices()
3
4      self._iniciar_relax()
```

```
5
6       for i in range(num_vertices-1):
7           for j in range(num_vertices):
8               for v, w in self.m.get_adyacentes(j):
9                   self._relax(j, v, w)
10
11      for i in range(num_vertices):
12          for v, w in self.m.get_adyacentes(i):
13              if self.d[v] > self.d[i] + w:
14                  raise Exception('Ciclo negativo detectado')
```

Visto el código, podemos obtener ahora un profile para analizar un pequeño grafo ejecutando el script *Profile.py* del apéndice C con el siguiente comando:

```
1  $python3.6 Profile.py
```

De esta forma obtendremos sus estadísticas de forma detallada y comprobaremos cual es la función que más tiempo consume. Por motivos prácticos se limitará la salida, pues serán las primeras líneas las que correspondan a las llamadas a funciones que más tiempo consumen y por ello las más interesantes de cara a optimizar el código en Cython.

Figura III.2: Salida del profile obtenido del algoritmo Bellman-Ford.

```
         864281499 function calls (864280673 primitive calls) in 258.314 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
861384812  148.688    0.000  148.688    0.000 BellmanFord.py:67(_relax)
        1  108.630  108.630  257.602  257.602 BellmanFord.py:77(ejecutar)
        1    0.559    0.559    0.594    0.594 BellmanFord.py:30(_convert_to_dic)
```

En la Figura III.2 se ve cómo el método `_relax` es el que más llamadas recibe (cosa obvia al encontrarse en la parte interna de las iteraciones), junto al método `ejecutar`, que se ejecuta una vez pero al ser el punto de acceso del algoritmo el tiempo total es importante. Es ahí, en estos dos métodos, donde habrá que hacer hincapié en nuestra optimización con Cython. El método `_convert_to_dic` corresponde a la clase `MatrizAdyacencia` y se encarga de la creación de un diccionario dada una matriz; sin embargo a efectos de este Trabajo de Fin de Grado, los tiempos para esta clase han quedado fuera, poniendo el foco en el tiempo que consumen los propios algoritmos dinámicos y no las operaciones previas como la generación de matrices aleatorias o la conversión automática de estas matrices a diccionarios que se realiza una vez cuando una clase `MatrizAdyacencia` se instancia o cuando se hace uso del método `set_obj_matriz`. De modo que estos tiempos, no repercuten en los resultados finales mostrados en este trabajo.

Pero antes de que empezar a trabajar en la optimización con Cython en estos dos métodos (`_relax` y `ejecutar`) existe otra forma más sencilla para ver la velocidad y el potencial de Cython sin tener que modificar una sóla línea de nuestro código Python puro,

esto es, renombrando el fichero **BellmanFord.py** que contiene el código de Python puro a **BellmanFord.pyx** y compilar el código utilizando el script en el fichero **setup.py** donde el contenido de este script se puede analizar en el Código III.7. La forma de ejecutar este script es la siguiente:

```
$python3.6 setup.py build_ext --inplace
```

*Código III.7: Fichero **setup.py** para la compilación de Cython.*

```
# setup.py
from distutils.core import setup, Extension
from Cython.Build import cythonize
import numpy

setup(
    ext_modules = cythonize(["BellmanFord.pyx"], annotate=True),
    include_dirs=[numpy.get_include()]
)
```

1.2.3. Código en Cython con tipos y directivas del compilador

Se van a aplicar las directivas `cython.boundscheck` y `cython.wraparound` para omitir tanto las comprobaciones de que nos encontramos dentro de las posiciones de memoria permitidas como las comprobaciones de índices negativos. Al hacer esto, se podría comprobar en la salida *HTML* cómo se reduce considerablemente el uso de llamadas a la API C/Python.

Se comenzó tipando los datos que se necesitan para la clase `BellmanFord` como se puede observar en el Código III.8, esto es, la instancia de un objeto `MatrizAdyacencia`, el vértice origen (s) es ahora es un `int`, la lista de distancias ahora es un puntero a `float` para posteriormente hacer la conveniente reserva de memoria con `malloc` y la lista de previos con `list`.

Código III.8: Variables en Bellman-Ford.

```
cdef class BellmanFord:
    nombre = 'Bellman-Ford'
    cdef public MatrizAdyacencia m
    cdef int s
    cdef float *d
    cdef list p
```

Pasamos ahora a analizar el constructor de la clase de Bellman-Ford mostrado en el Código III.9. Como buena práctica de programación, se inicializan las variables estableciendo el puntero a `float` a `NULL`, el nodo *source* (llamado `s`) a `s` que ahora se encuentra tipado a tipo `int` y la lista de previos `p` a una lista vacia. Se ha tipado (en la línea 6) el parámetro `m` del prototipo del método `set_obj_matriz`, que establece una matriz a nuestra elección, a tipo `MatrizAdyacencia`.

Código III.9: Constructor de Bellman-Ford en Cython.

```
1       def __cinit__(self, int s):
2           self.s = s
3           self.d = NULL
4           self.p = []
5
6       def set_obj_matriz(self, MatrizAdyacencia m):
7           self.m = m
```

Los cambios realizados en la inicialización del algoritmo mostrado en el Código III.10, han sido tipar el número de vértices en un tipo `int` y el índice `i` con tipo `Py_ssize_t` lo que producirá un bucle puro en C, como ya se vió anteriormente.

Código III.10: Inicialización de Bellman-Ford en Cython.

```
1       @cython.boundscheck(False)
2       @cython.wraparound(False)
3       cdef _iniciar_relax(self):
4           cdef int num_vertices = self.m.m.shape[0]
5           cdef Py_ssize_t i
6           for i in range(num_vertices):
7               self.d[i] = np.inf
8               self.p.append(None)
9           self.d[self.s] = 0
```

En el Código III.11 se han tipado los parámetros para la indexación a tipo `Py_ssize_t` y el coste a un tipo `float`. Se ha eliminado las variables locales para obtener los resultados directamente.

Código III.11: Operacion de relajamiento de Bellman-Ford en Cython.

```
1       @cython.boundscheck(False)
2       @cython.wraparound(False)
3       cdef _relax(self, Py_ssize_t u, Py_ssize_t v, float w):
4           if self.d[v] > self.d[u] + w:
5               self.d[v] = self.d[u] + w
6               self.p[v] = u
```

Por último, el método `ejecutar` se muestra en el Código III.12. En su versión pura de Python mostrada en el Código III.6 se vio que contiene un importante número de de bucles. Todos estos bucles tienen ahora sus respectivos índices, los cuales, han sido convertidos a tipo `Py_ssize_t`. El coste `w` utilizado para la operación de relajamiento, es ahora un tipo `float` y finalmente el número de vértices `num_vertices` es un tipo `Py_ssize_t`. Es en este método donde se ha realizado la reserva de memoria dinámica en el puntero de las distancias con el debido casting en Cython. Siempre que se hace una reserva de memoria es muy importante (al igual que en C), comprobar que no habido un error (que devolvería NULL). Para ello, comprobamos que la variable no es `NULL` y en caso de que así lo fuera se lanza la excepción `MemoryError`. El código restante no cambia, con la salvedad que al final del método se debe liberar la memoria en caso de que el puntero no sea `NULL` con la función de la librería estandar de C `free` como se aprecia en la línea 27.

Código III.12: Ejecutar de Bellman-Ford en Cython.

```
@cython.boundscheck(False)
@cython.wraparound(False)
cdef ejecutar(self):
    cdef Py_ssize_t i, j
    cdef Py_ssize_t u, v
    cdef float w = 0
    cdef Py_ssize_t num_vertices

    self.d = <float*>malloc(self.m.m.shape[0]*sizeof(float))

    if not self.d:
        raise MemoryError()

    self._iniciar_relax()

    num_vertices = self.m.m.shape[0]
    for i in range(num_vertices-1):
        for j in range(num_vertices):
            for v, w in self.m.dic.get(j):
                self._relax(j, v, w)

    for i in range(num_vertices):
        for v, w in self.m.dic.get(i):
            if self.d[v] > self.d[i] + w:
                raise Exception('Ciclo negativo detectado')

    if self.d != NULL:
        free(self.d)
```

1.3. Cálculo de tiempos

Este cálculo de tiempos para el algoritmo de Bellman-Ford se ha obtenido utilizando un procesador *Intel® Core™ i7-8700K CPU @ 3.70GHzx12* con la distribución GNU/Linux Ubuntu 18.04.3 LTS.

En la Figura III.3 se refleja los tiempos para una serie de grafos (tiempo frente al número de vértices). Para cada grafo, se ha calculado el tiempo de ejecución con tres tipos de código: uno en Python puro, otro compilado directamente a través de Cython y el último en Cython. Para el algoritmo de Bellman-Ford los grafos han tenido un intervalo entre 500 vértices y 2.500 vértices, esto es, se ha hecho uso del método `ejecutar_aleatorias` anteriormente descrito para que genere grafos desde 500 vértices hasta 2.500 vértices en incrementos de 500. Una vez guardados estos grafos, se ejecuta el método `obtener_ficheros_serializados` de la clase `SerializarMatrices` para que lea la información de esta carpeta pero sin generar nuevos grafos aleatorios; de esta forma, los grafos son los mismos para los tres tipos de código.

Solo con la compilación de código Python a través de Cython se obtiene ya una significativa mejora. El aplicar un estudio de profile reveló de forma inequívoca que la función con el tiempo de ejecución más alto en el algoritmo de Bellman-Ford es la operación de relajamiento `_relax` junto el método `ejecutar`; con esta información, aplicando un tipado éstatico junto a la desactivación de algunas directivas como `cython.boundscheck`

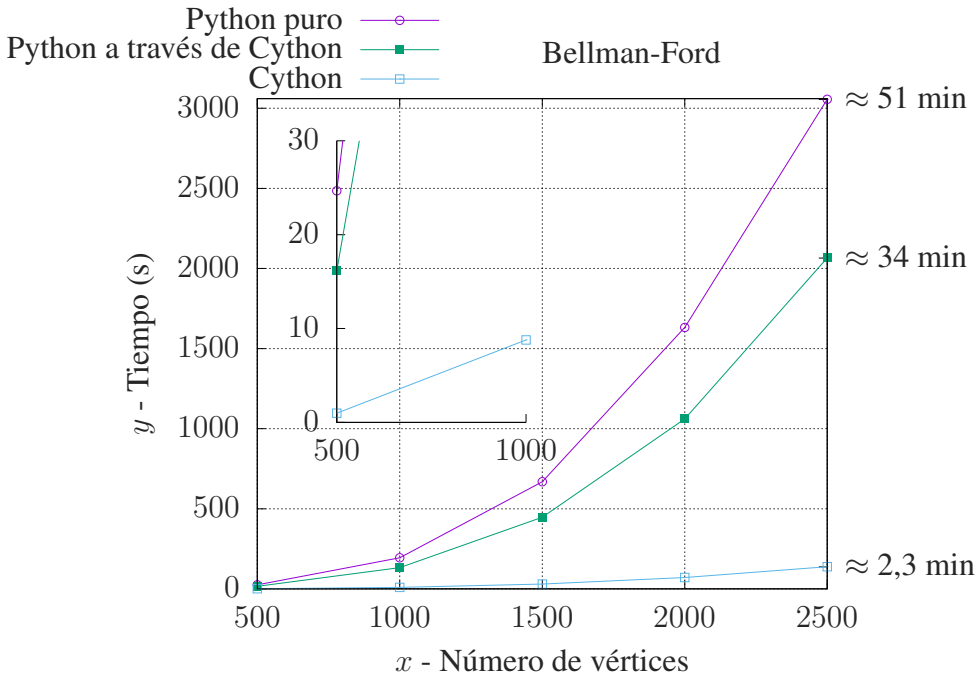


Figura III.3: Cálculo de tiempos del algoritmo Bellman-Ford. Se ha ampliado la escala para el grafo con 500 *vértices. Se ve una muy buena mejora con el uso de Cython para un grafo de* 2.500 *vértices. Todo ello gracias a un tipado estático de las variables, uso de memoria dinámica y directivas del compilador.*

y `cython.wraparound`, podremos reducir el tiempo de forma relevante.

Importante señalar que este cálculo de tiempos se obtuvo sin incluir la ejecución de profile. El uso de éste a la hora de obtener los tiempos podría repercutir de forma muy negativa en la velocidad debido al *overhead*; por ejemplo, en la versión de Python puro se comprobó de forma específica que para el grafo de 2.500 vértices, hubo un incremento de alrededor de 17 minutos al haber ejecutado el algoritmo usando el profile. Por tanto, cuando se requiera conocer qué partes de un código son las que más tiempo consumen, no nos dejaremos llevar por nuestra intuición, sino que primero aplicaremos nuestro profile sobre el código Python base, luego aplicaremos Cython, y finalmente calcularemos tiempos sin el uso de profile.

A continuación se pasa a explicar el algoritmo de Ford-Fulkerson, su pseudocódigo/código y, posteriormente la conversión a Cython y con ello la obtención de tiempos.

2. Algoritmo de Ford-Fulkerson

El algoritmo Ford-Fulkerson se encuentra en la teoría de `redes de flujo` y presenta tres importantes ideas para muchos algoritmos de flujo: las redes residuales, el aumento de flujo a lo largo un camino y los cortes (cuts). Se le suele llamar el método de Ford-

Fulkerson, pues éste es la base para otros algoritmos que aplican sus principios. En este tipo de algoritmos, hay un producto que "fluye" desde un origen (donde es producido) a un sumidero (donde es consumido).

2.1. Redes de flujo

Los algoritmos de redes de flujo resuelven una amplia cantidad de problemas, entre los cuales encontramos *problemas de distribución*, donde un tipo de inventario se mueve de un punto a otro dentro de la red o *problemas de tráfico*, donde por ejemplo requiere obtener el mínimo de tiempo para evacuar una ciudad suponiendo que el tráfico fuera un flujo.

Una red de flujo se define como un grafo dirigido $G = (V, E)$, donde en cada arista $(u, v) \in E$ tiene una capacidad no negativa $c(u, v) \geq 0$ y $c(u, v) = 0$ si $(u, v) \notin E$. En el grafo, se pueden distinguir dos vértices destacados que denotamos como el origen *s* (de source) y el sumidero *t* (de target). Un *flujo* en G es una función real de $f : V \times V \to \mathbb{R}$ y que satisfará tres propiedades:

1. **Antisimetría:** $\forall u, v,\ f(u, v) = -f(v, u)$
2. **La restricción de la capacidad:** para todos los vértices $(u, v) \in V$, se require $f(u, v) \leq c(u, v)$, es decir, que el flujo de una arista nunca será mayor que la capacidad fijada para ésta.
3. **La conservación del flujo:** $\forall u \in V - \{s, t\}$, es decir, para todos los vértices del grafo excluyendo el origen s y el sumidero t, se requiere:

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v) \tag{3}$$

esto es, que el flujo total que entra en cada vértice es igual al flujo total del saliente. Esto es el equivalente a la ley de Kirchhoff cuando el material es corriente eléctrica.

Si $(u, v) \notin E$, entonces no puede haber flujo de u a v y según la función f anteriormente definida tendremos $f(u, v) = 0$.

El primer contacto con el algoritmo de Ford-Fulkerson se puede tener en el Código III.13 y es la base para otros algoritmos de flujo. También se ven términos como rutas de aumento o redes residuales.

Código III.13: Pseudocódigo para el método de Ford-Fulkerson.

```
1  FORD-FULKERSON-MÉTODO(G, s, t)
2      inicializar el flujo f a 0
3      Mientras haya una ruta de aumento p desde s hasta t en la red residual G_f
```

```
4           actualizar el flujo f a lo largo de p
5       return f
```

2.2. Redes residuales

En un grafo G y un flujo f, la red residual $G_f = (V, E_f)$ consiste en aristas con capacidades que muestran cómo podemos cambiar el flujo en las aristas de G. En cada arista de la red se puede admitir un cantidad de flujo adicional igual a la capacidad de la arista menos el flujo de esa arista. Si el valor es positivo, ponemos esa arista dentro de E_f con una capacidad residual dada por $c_f(u, v) = c(u, v) - f(u, v)$. Las únicas aristas de G que se encuentran en G_f son aquellas que pueden admitir más flujo las que tienen un flujo máximo (el valor máximo de la capacidad) cumplen $c_f(u, v) = 0$ y por tanto no están en G_f.

Se podría decir que los grafos residuales son imagenes temporales de los cambios (incrementos y/o decrementos) que se van actualizando en los flujos; es por eso, que podamos terminar con aristas en G_f que no están en el grafo G. Como el algoritmo quiere buscar el máximo flujo, es posible que tenga que decrementar el flujo de un arista para llevarlo a otra. Esto se traduce en una capacidad residual $c_f(u, v)$ con la ecuación (4).

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{si } (u,v) \in E \\ c(v,u) - f(v,u) & \text{si } (u,v) \notin E \end{cases} \tag{4}$$

Imaginemos que tenemos un par de vértices u y v –como se observa en la Figura III.4– con una capacidad en la arista de $c(u, v) = 7$ y un flujo que pasa por ésta en ese momento de $f(u, v) = 4$ unidades. Tendremos entonces una capacidad máxima residual de $c_f(u, v) = 7 - 4 = 3$ y se podrá incrementar un nuevo flujo $f(u, v)$ hasta alcanzar esa capacidad. De igual forma podremos mover flujo en sentido opuesto, esto es, decrementando el flujo y su capacidad residual será $c_f(v, u) = f(u, v) = 4$.

2.3. Aumento de flujo a lo largo un camino

Si en una red residual G_f tenemos un camino p desde el vértice s (origen) a t (destino), se puede ir aumentando el flujo a lo largo del camino hasta el máximo de $c_f(u, v)$. Para ello, aumentamos el flujo en cada arista del grafo residual con un máximo de la capacidad mínima que hayamos encontrado en ese camino. Este mínimo viene dado por la ecuación (5).

$$c_f(p) = min\{c_f(u,v) : (u,v) \text{ se encuentra en } p\} \tag{5}$$

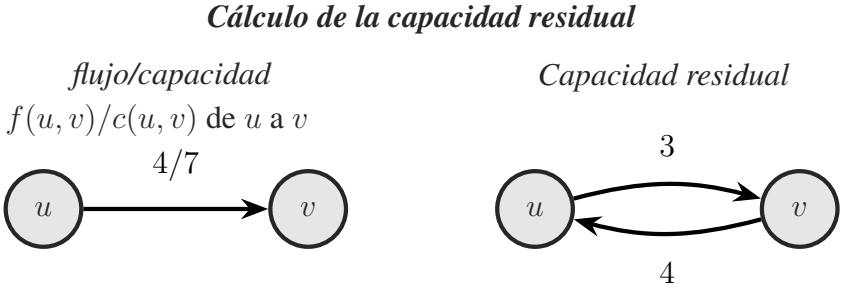


Figura III.4: Proceso para el cálculo de la capacidad residual. Importante señalar que / no es la operación de división. Con ese símbolo separamos el número que representa el flujo (a la izquierda) con la capacidad (a la derecha). La capacidad es el "tope" de flujo que puede pasar por una arista. A la derecha, el par de vértices mostrando el cálculo de la capacidad residual resultante junto con el flujo que puede admitir en sentido contrario.

2.4. Pseudocódigo y código

A continuación se véra el pseudocódigo de las partes del algoritmo Ford-Fulkerson para después proceder a ver el código en Python puro junto a su transformación en Cython. Una pregunta que uno se podría hacer es cómo obtener las rutas de aumento; esto es sencillo, bastaría con usar el algoritmo *búsqueda en anchura* (Breadth-First Search o BFS) o el de *búsqueda en profundidad* (Depth-First Search) sobre G_f. Se estudiará en el pseudocódigo el *BFS*.

2.4.1. Pseudocódigo

El pseudocódigo comienza inicializando el flujo a cero en todas las aristas del grafo. Luego en cada iteración se va incrementando el valor del flujo en G mientras se vaya encontrando una ruta de aumento en un grafo residual G_f.

El método de Ford-Fulkerson visto en el Código III.13 es una versión resumida del algoritmo Ford-Fulkerson; si este método lo expandimos obtenemos el pseudocódigo del Código III.14

Código III.14: Pseudocódigo para el algoritmo Ford-Fulkerson.

```
1  FORD-FULKERSON(G, s, t)
2      Para cada arista (u,v) ∈ G.E
3          (u,v).f = 0
4
```

```
5       G_f = G
6       Mientras haya una ruta de aumento p desde s hasta t en la red residual G_f
7           c_f(p) = min{c_f(u,v) : (u,v) está en p}
8
9           Para cada arista (u,v) en p
10              si (u,v) ∈ E
11                  (u,v).f = (u,v).f + c_f(p)
12              si no
13                  (v,u).f = (v,u).f - c_f(p)
14      return TRUE
```

Las rutas de aumento p desde s a t se encuentran habiendo hecho uso del algoritmo de búsqueda en anchura en G_f con un vértice origen s. Aclarar que este algoritmo explora el grafo, pero sin un objetivo t en particular; sólo partirá de un origen y desde ahí iremos construyendo la lista de previos. Por tanto, si queremos encontrar un camino con la ayuda de BFS, deberemos haber ya explorado de antemano el grafo, obtenido la tabla de previos y, dado un objetivo t, deberemos hacer un *backtracking* (esto es, ir hacia atrás) en la búsqueda del nodo origen s. Si lo encontramos, existirá un camino; si no (encontramos un `NULL/None`), no habrá camino posible desde s a t. El pseudocódigo se puede observar en el Código III.15 y el código completo se encuentra implementado en el método `bfs` de la clase `FordFulkerson` en su versión Python pura en el apéndice G o bien, en su versión Cython en el apéndice H.

Código III.15: Pseudocódigo para la búsqueda en anchura de un grafo.

```
1   BREADTH-FIRST-SEARCH(G, s)
2       para cada vértice u ∈ G.V - {s}
3           u.color = BLANCO
4           u.d = ∞
5           u.p = NULL
6
7       s.color = GRIS
8       s.d = 0
9       s.p = NULL
10      Q = ∅
11      ENCOLAR(Q, s)
12      Mientras Q ≠ ∅
13          u = SACAR_DE_LA_COLA(Q)
14
15          para cada vértice v ∈ G.Adj[u]
16              if v.color == BLANCO
17                  v.color = GRIS
18                  v.d = u.d + 1
19                  v.p = u
20                  ENCOLAR(Q, v)
21          u.color = NEGRO
```

2.4.2. Código en Python puro

Se ha elegido un pequeño grafo (1.000 vértices) para ejecutar un profile (de nuevo limitando su sálida) del código puro en Python; de esa forma nos podremos hacer una idea de las funciones que provocan una lenta ejecución.

De forma empírica, el método `bfs`, responsable de buscar las rutas de aumento, es

Figura III.5: Salida del profile obtenido del algoritmo Ford-Fulkerson.

```
        51357120 function calls (51356294 primitive calls) in 61.634 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1870   50.069    0.027   61.051    0.033 FordFulkerson.py:88(bfs)
  1869001    1.878    0.000    5.036    0.000 queue.py:147(get)
  1869001    1.558    0.000    3.869    0.000 queue.py:115(put)
```

una de las partes del código que más repercute en la velocidad. Es en éste método donde se utiliza las **colas FIFO** de Python y en el que se hace un uso exhaustivo del método **get**, para obtener el último elemento de la cola y **put**, para añadirlo. En Python puro, las colas FIFO utilizan objetos Python (incluso para los enteros), lo que lleva a un gasto de tiempo extra.

El procedimiento para la compilación a través de Cython es el mismo que el mostrado anteriormente para el algoritmo Bellman-Ford, esto es, cambiar la extensión del fichero con el código de Python puro **FordFulkerson.py** a su respectiva extensión de Cython **FordFulkerson.pyx**. Ahora, con esta extensión, podremos sin problemas compilarlo con el fichero **setup.py** mostrado anteriormente en el Código III.7.

2.4.3. Código en Cython con tipos, directivas del compilador y librerias externas

De igual forma que se vio en el algoritmo de Bellman-Ford se han aplicado las directivas `cython.boundscheck` y `cython.wraparound`.

Se ha tipado en el Código III.16 la matriz de adyacencia con `MatrizAdyacencia` y se ha puesto la palabra reservada `public` para poder acceder desde fuera a estas instancias evitando overheads de métodos *getters*.

Código III.16: Variables en Ford-Fulkerson en Cython.

```
1  cdef class FordFulkerson:
2      nombre = 'Ford-Fulkerson'
3      cdef public MatrizAdyacencia m
4
5      def set_obj_matriz(self, MatrizAdyacencia m):
6          self.m = m
```

En el método `ejecutar`, se han tipado las variables locales como `m_flow` y `min_cap` junto a los índices `prev` y `t_tmp` para una rápida indexación de la matriz. Se ha reservado memoria dinámica para la tabla de `previos`, realizando el conveniente casting a `<int*>` (puntero a entero) y su correcta liberación.

Código III.17: Ejecutar de Ford-Fulkerson en Cython.

```
1  cdef ejecutar(self, int s, int t):
```

```
2       cdef float m_flow = 0
3       cdef float min_cap = 0
4       cdef Py_ssize_t prev, t_tmp
5       cdef int *previos = <int*>malloc(self.m.c_np.shape[0]*sizeof(int))
6
7       if not previos:
8           raise MemoryError()
9
10      while self.bfs(s, t, self.m, previos):
11          min_cap = self._calcular_minimo(s, t, previos)
12          m_flow += min_cap
13          t_tmp = t
14          while s != t_tmp:
15              prev = previos[t_tmp]
16              self.m.c_np[t_tmp,prev] += min_cap
17              self.m.c_np[prev,t_tmp] -= min_cap
18              t_tmp = prev
19
20      if previos != NULL:
21          free(previos)
22      print("Flow maximo: {}".format(m_flow))
```

En el método `_calcular_minimo` mostrado en el Código III.18, se han tipado los parámetros de los vértices `s` y `t` a **`int`** y la lista de `previos` a puntero a **`float`**. La variable local `min_cap` ha sido tipada a **`float`** y las variables para indexar en la matriz `prev` y `t_tmp` han sido tipadas como `Py_ssize_t`.

Código III.18: Cálculo del mínimo en Ford-Fulkerson en Cython.

```
1   cdef _calcular_minimo(self, int s, int t, int *previos):
2       cdef Py_ssize_t t_tmp = t
3       cdef float min_cap = np.inf
4
5       while s != t_tmp:
6           min_cap = min(min_cap, self.m.c_np[previos[t_tmp],t_tmp])
7           t_tmp = previos[t_tmp]
8       return min_cap
```

Finalmente y sugerido por el profile realizado anteriormente, el método `bfs` se muestra en el Código III.19. Aquí habrá una magnífica oportunidad para utilizar la primera librería externa en **puro C** junto a Cython. Para ello, se partió de la **librería de algoritmos en C** (en especial el **tipo de dato abstracto** (TAD) *Queue* para este Trabajo de Fin de Grado) desarrollada por Simon Howard con licencia ISC y recomendada por la documentación oficial de Cython. Esta librería se realizó en estandard ANSI[2] donde encontraremos implementaciones como *tablas Hash*, *colas*, *listas enlazadas simples* y *dobles*, etcétera. Para el método `bfs` se ha hecho uso de la implementación de *Queue* o *cola*. Howard tiene una web [13] donde se puede encontrar tanto la documentación como la dirección a su repositorio en GitHub. Se aclara que cada *TAD* no necesita de otras librerías externas, por lo que su finalidad es usarlos de forma independiente según las necesidades.

En referencia al Código III.19, la implementación de la búsqueda en anchura es sencilla; sin embargo, hay que tener en cuenta algún que otro detalle para la correcta

[2]También llamado "ANSI C".

implementación con el algoritmo de Ford-Fulkerson. Como cualquier búsqueda, recibe como argumentos el nodo origen (*s*) y el destino (*t*), la propia información del grafo dada por `m` (una `MatrizAdyacencia`) y un puntero al array de `previos`. Se han tipado todos los índices que se usarán a lo largo del método como son `i`, `u` y `v`. La lista `vistos` ahora está reservada de forma dinámica; este array devuelve `True` o `False` según se haya encontrado o no el destino en este array de `vistos`. Si hubiera algún tipo de error en la reserva con `malloc` se devolvería una excepción del tipo `MemoryError`.

En la línea 12 se hace uso de la función `memset` para establecer a cero el espacio reservado con memoria dinámica para el array de `vistos`. En la línea 14 se crea una cola haciendo uso de la librería en C puro de Howard. De igual forma que con cualquier reserva de memoria, se hacen las oportunas comprobaciones en la línea 16.

Se pone a uno el vértice origen y seguidamente lo insertamos en la cola FIFO. Iteramos mientras la cola no esté vacía y vamos expandiendo los nodos desde la línea 22 a la 27. La primera operación es sacar el último elemento de la cola –nuestro vértice origen–, después obtenemos sus nodos adyacentes junto con la capacidad hasta ellos y vamos construyendo el camino desde ellos si han sido vistos o no, para explorarlos y para guardar desde qué vértice llegamos a ellos, esto es, el previo. La condición incluye la **capacidad**, pues debe ser positiva desde u a v. Si las dos condiciones se cumplen (el vértice no ha sido visitado y la capacidad no es cero) entonces marcamos el vértice actual como visto, guardamos el nodo con el que llegamos a él (previo) y añadimos a la cola el vértice actual para su posterior exploración.

Código III.19: Búsqueda en anchura para Ford-Fulkerson en Cython.

```
1  @cython.boundscheck(False)
2  @cython.wraparound(False)
3  cdef bfs(self, int s, int t, MatrizAdyacencia m, int *previos):
4      cdef Py_ssize_t i, u, v
5      cdef float w = 0.0
6      cdef int *vistos = <int*>malloc(m.c_np.shape[0]*sizeof(int))
7      cdef int esta_visto = 0
8
9      if not vistos:
10         raise MemoryError()
11
12     memset(vistos, 0, sizeof(int)*m.c_np.shape[0])
13
14     cdef cqueue.Queue *q = cqueue.queue_new()
15
16     if not q:
17         raise MemoryError()
18
19     vistos[s] = 1
20     cqueue.queue_push_tail(q, <int*>s)
21     while not cqueue.queue_is_empty(q):
22         u = <int>cqueue.queue_pop_head(q)
23         for v in m.dic.get(u):
24             if vistos[v] == False and m.c_np[u][v] > 0:
25                 vistos[v] = 1
26                 previos[v] = u
27                 cqueue.queue_push_tail(q, <int*>v)
28
29     esta_visto = vistos[t]
```

```
30
31      if q != NULL:
32          cqueue.queue_free(q)
33
34      if vistos != NULL:
35          free(vistos)
36
37      if esta_visto:
38          return True
39      return False
```

Para hacer uso de las funciones de las librerías externas, necesitaremos redefinir[3] (al menos las que vayamos a utilizar de la implementación pura en C) estas declaraciones del fichero de cabecera en C **queue.h** en el fichero de cabecera de Cython **cqueue.pxd**. En el fichero pxd, mostrado en el Código III.20, tendríamos a disposición toda la implementación completa. Para el algoritmo Ford-Fulkerson se usaron `queue_new`, para su reserva de memoria, `queue_free`, para su liberación, y las funciones `queue_push_tail` y `queue_pop_head` para su insercción y extracción respectivamente.

Código III.20: Fichero de cabecera para Ford-Fulkerson en Cython.

```
1   # cqueue.pxd
2
3   cdef extern from "queue.h":
4       ctypedef struct Queue:
5           pass
6       ctypedef void* QueueValue
7
8       Queue* queue_new()
9       void queue_free(Queue* queue)
10
11      int queue_push_head(Queue* queue, QueueValue data)
12      QueueValue  queue_pop_head(Queue* queue)
13      QueueValue queue_peek_head(Queue* queue)
14
15      int queue_push_tail(Queue* queue, QueueValue data)
16      QueueValue queue_pop_tail(Queue* queue)
17      QueueValue queue_peek_tail(Queue* queue)
18
19      bint queue_is_empty(Queue* queue)
```

Ahora se debe hacer un linkado, ya sea estático o dinámico. Para hacer uno estático bastará incluir como primera línea en nuestro fichero Cython con extensión **pyx** la siguiente directiva de compilador:

```
1   # distutils: sources = queue.c
```

2.5. Cálculo de tiempos

Para este cálculo de tiempos se ha utilizado de nuevo un procesador *Intel® Core™ i7-8700K CPU @ 3.70GHzx12* con la distribución GNU/Linux Ubuntu 18.04.3 LTS.

[3] Una posible mejora en Cython es que no se tenga que volver a declarar las funciones.

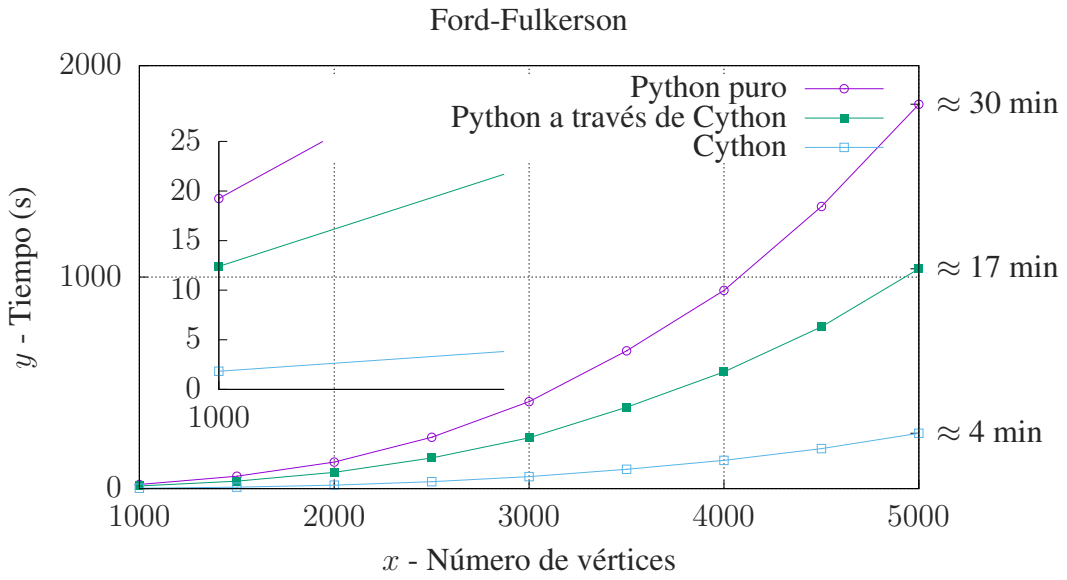


Figura III.6: Cálculo de tiempos de Ford-Fulkerson. Se obtuvo en la versión de Cython una aceleración 7 veces más respecto a su versión Python puro gracias al tipado, al uso de memoria dinámica y a la utilización de una librería externa de colas realizada en C.

En la Figura III.6 se adjunta los resultados de la ejecución del algóritmo Ford-Fulkerson, de nuevo, número de vértices frente al tiempo. Para todos los grafos se ha obtenido el tiempo con una versión pura en Python, otra compilando a través de Cython y finalmente otra en Cython. El tamaño de los grafos han sido entre 1000 y 5.000 vértices en incrementos de 500 nodos. Se ha impuesto un valor máximo de 1000 unidades para la capacidad generada de forma aleatoria. El factor de densidad se ha establecido con un valor del $90\,\%$. Se puede observar que en la versión de Python a través de Cython (sin realizar ningún cambio) redujo el tiempo casi a la mitad, y finalmente con un uso de librerías externas de C, directivas de compilación, tipados y memoria dinámica, se ha podido reducir unos 26 minutos.

La mejora que obtenemos aplicando tipos, índices, directivas de compilador en Cython y el uso de librerías externas (`cola` y `memset`) es buena pero quizás no tanto como cabría esperar si se compara con Bellman-Ford. La salida de la ejecución del profile ya sugería, que la función `bfs` sería la candidata para trabajar más adelante con Cython; pues hacia uso de librerías en Python puro, como las colas. Fue por esto, que se movió estas llamadas en Python puro a librerías externas realizadas en código C puro para esta tarea y la reserva de memoria dinámica para la tabla de `vistos`. Ahora, realizado esto y fuera de dudas de que el mal rendimiento no puede ser debido a un *overhead* en Python, la posible razón de no obtener la mejora esperada, puede ser debida a la propia naturaleza del algoritmo de Ford-Fulkerson con su variedad de diferentes formas de tratar la información en los bucles y donde el uso escaso de índices a la hora de obtener las

rutas de aumento p finalmente anulan la mejora que se podría obtener en la sección del código para la actualización del flujo y la obtención del mínimo una vez calculada una ruta de aumento.

Por ello, con la eficiente optimización que produce Cython de forma nativa a través de Python sería suficiente para algoritmos de flujo, pues todos ellos, hacen uso del método de Ford-Fulkerson.

3. Conclusiones

Este capítulo no sólo ha servido para conocer el estudio de dos importantes algoritmos en Ciencias de la Computación sino también para comprobar de forma empírica cómo el uso de **Cython** en problemas donde haya varias iteraciones puede acelerar de una forma ventajosa nuestros algoritmos sin el conocimiento y la experiencia que implica el desarrollo de módulos de Python en lenguaje C. Se ha observado, que en situaciones en que haya muchos bucles vale la pena el esfuerzo de optimizar con índices junto al tipado en argumentos de funciones y desactivando cualquier tipo de comprobación de errores a través de las directivas del compilador; con ello se conseguirá de forma notoria un incremento en velocidad. Si tenemos listas Python (donde podremos guardar cualquier tipo de objeto) generalmente contendrán el mismo tipo de objeto o números; es por eso, que será necesario hacer uso de memoria dinámica para crear un array con un casting a puntero de nuestro tipo. Será muy recomendable ver si es posible cambiar implementaciones de código Python puro como *colas* o *tablas Hash* u otros tipos a su versión pura de C. Todos estos pequeños cambios en su conjunto, harán que incremente la velocidad de forma drástica. Una de las grandes ventajas de Cython es que según ahondemos más en su aprendizaje (memoria dinámica, librerías externas...) podremos reducir más el tiempo y siempre en un entorno flexible y sencillo como es el lenguaje Python.

En el siguiente capítulo se ahondará en el estudio de ecuaciones diferenciales ordinarias (*EDOs*) junto a los métodos más comunes para su resolución. Después, con un conocimiento más profundo de ellas, se desarrollarán algoritmos en Python para acto seguido poner en práctica de nuevo Cython y observar si puede ser una buena herramienta para la resolución de EDOs.

Capítulo IV

Aplicación de Cython en ecuaciones diferenciales

> I understand what an equation means if I have a way of figuring out the characteristics of its solution without actually solving it.
>
> *Paul Dirac*

El objetivo principal de este capítulo es hacer un estudio de las ecuaciones diferenciales (*ED*), con foco en las ordinarias, con el fin de comprenderlas mejor con vistas a su resolución numérica. Explicar de forma teórica los métodos más comunes para su resolución será una parte clave en este capítulo; de esta manera se reconocerá la idea principal que subyace detrás del algoritmo para después realizar su implementación en Python y acto seguido usar Cython.

Acerca de su definición, una bonita forma sería decir que son una herramienta con la que podemos predecir el futuro; ya sea una posición de los astros en los cielos [1] o el clásico ejemplo de crecimiento de una población. Y de forma algo más analitica, que si en el álgebra se busca números desconocidos que satisfagan una ecuación, en una ecuación diferencial lo que buscamos es una función desconocida que estará relacionada con una o más de sus derivadas.

Aunque se verá a continuación cómo clasificarlas y qué significa el orden, se pueden expresar por la **forma general** una ecuación ordinaria de orden n con una variable dependiente $y = y(x)$ y variable independiente x, donde F indica una función de valores reales de $n + 2$ variables como sigue:

$$F\Big(x, y, y', ..., y^{(n)}\Big) = 0 \tag{1}$$

Se puede observar en la ecuación (1) que se ha hecho uso de la **notación prima** $y', y'', y''', ...$ para las derivadas, de esta forma las ecuaciones son más compactas aunque con la desventaja de que esconde tanto las variables dependientes como las independientes. Otra forma hubiera sido representar la ecuación (1) usando la **notación de Leibniz** $dy/dx, d^2y/dx^2, d^3y/dx^3, ...$ de la siguiente forma:

$$F\Big(x, y, \frac{dy}{dx}, ..., \frac{dy^n}{dx^n}\Big) = 0 \tag{2}$$

[1] En el siguiente intervalo infinitesimal de tiempo; es decir, una variable dependiente –el espacio– con otra independiente –el tiempo–.

Y en la **forma normal**:

$$\frac{d^n y}{dx^n} = f\left(x, y, y', ..., y^{(n-1)}\right) \tag{3}$$

f al igual que F son funciones continuas con valores reales.

1. Clasificación

El primer paso para estudiarlas es conocer su clasificación segun sea su *tipo*, su *orden* y su *linealidad*. Cada uno de estos puntos de clasificación se ve a continuación.

1.1. Tipo

Una ED puede ser *ordinaria* (EDO) o *parcial* (EDP). Las primeras ecuaciones incluyen derivadas de una o más variables dependientes respecto a **una sola** variable **independiente**. Un ejemplo de éstas podrían ser las siguientes:

$$\frac{dy}{dx} + 9y = e^x, \qquad \frac{d^2y}{dx^2} - \frac{dy}{dx} + 2y = 0, \qquad \frac{dx}{dt} + \frac{dy}{dt} = 3x + 2y \tag{4}$$

Las dos primeras tienen una variable dependiente y, pero la última tiene dos x e y y aun así ésta última pertenece también a esta categoria.

Por otro lado, si la variable dependiente es una función de dos o más variables independientes, entonces habrá derivadas parciales. Unos ejemplos son los siguientes:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \qquad \frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x} \tag{5}$$

1.2. Orden

Es el orden de la mayor (más alta) derivada en una ecuación; por ejemplo, en

$$\frac{d^2y}{dx^2} + 5\left(\frac{dy}{dx}\right)^3 - 7y = e^x \tag{6}$$

hay que fijarse que esta ecuación es de orden 2 (y no 3).

1.3. Linealidad

La ecuacion (1) será **lineal** si F es lineal en $y, y', ..., y^{(n)}$, es decir, cuando una ecuación diferencial ordinaria de orden n-ésimo es de la forma

$$a_n(x)\frac{d^n y}{dx^n} + a_{n-1}(x)\frac{d^{n-1} y}{dx^{n-1}} + ... + a_1(x)\frac{dy}{dx} + a_0(x)y = g(x) \tag{7}$$

En la ecuación (7) se observan dos importantes propiedades: primero, los coeficientes $a_0, a_1, ..., a_n$ de $y, y', ..., y^{(n)}$ dependen de x, que es la variable independiente. Y segundo, la variable dependiente y y sus derivadas $(y', y'', ..., y^{(n)})$ actúan como polinomios de primer grado. Si no se cumplen estas propiedades la ecuación es **no lineal**.

2. Uso de campos direccionales para su interpretación

Las ED de por sí son una rica fuente de información si las vemos de forma geométrica. Por ejemplo, si uno se fija en la siguiente ecuación:

$$\frac{dy}{dx} = f(x, y) \tag{8}$$

veremos que representa un campo de pendientes (o campo direccional); es decir, en todo punto (x, y) del plano xy la función $f(x, y)$ nos proporcionará una pendiente. La pendiente de una recta en cada punto nos sugerirá un método gráfico para obtener soluciones aproximadas a la ecuación diferencial vista en (8). O en otras palabras, el campo de pendientes, nos dará una idea visual de las soluciones, que todas ellas deberán ser tangentes a todos estos segmentos cortos por los que pasen.

Se podría dar valores a mano a las variables x e y e ir dibujando en papel todos estos elementos lineales, lo que al final resultaría en un arduo trabajo. Otra alternativa mejor es dejar esta tarea a las máquinas y usar una herramienta algebraica como es **GeoGebra**, un completísimo paquete matemático interactivo GPL. En la figura IV.1 se observa un campo de pendientes de la función $dy/dx = x - y$ generado con **GeoGebra**.

En el campo de pendientes se han incluido dos curvas solución. Se puede ver de forma visual que cuantos más segmentos de linea se incluyan, más precisas podrán ser las soluciones. De un simple vistazo en el campo de pendientes se ve que otra solución es la linea recta $y = x - 1$.

3. Métodos para su resolución numérica

En esta sección se verán algunos métodos de resolución numérica. Uno de ellos –el más sencillo– es el llamado el **método de Euler** que utiliza la tangente para ir obteniendo valores aproximados cercanos al punto de tangencia. Luego se verá las modificaciones del método de Euler para conseguir el **método de Euler mejorado** y por último se verá el método de los alemanes *Carl Runge* y *Wilhelm Kutta* bien conocido por sus apellidos como **Runge-Kutta**.

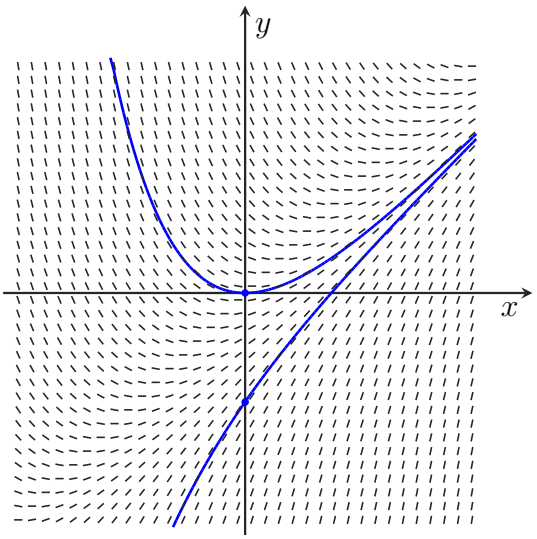


Figura IV.1: El campo de pendientes generado para la ecuación $dy/dx = x - y$. En situaciones físicas, el campo de isoclinas (o pendientes) nos proporcionará información de cómo se comporta el sistema que de otra forma, con la ecuación misma sería complicada.

3.1. El Método de Euler

El método de Euler hace uso de la **linealización**, como se puede ver en la Figura IV.2; es decir, aproximar la curva solución por una secuencia de segmentos rectos. Cada pequeño segmento tendrá una longitud horizontal h y su pendiente vendrá dada por el valor $f(x, y)$ justo en el extremo final del segmento anterior. La pendiente es directamente obtenida por la función $f(x, y) = y'$ dada por la ecuación diferencial. El tamaño de paso entre x_n y x_{n+1} es una constante h y es importante elegir un tamaño de paso que sea pequeño pues la bondad de nuestra aproximación dependerá de éste. Pero al hacer esto se requerirán más pasos $\left(n = \dfrac{|x - x_0|}{h}\right)$ para llegar desde x_0 al valor x donde queramos conocer el valor de la solución. Si nos fijamos en la Figura IV.2(a), f varía a lo largo del segmento y el error producido por la discretización vendrá dado por el valor de $f(x, y)$ en un extremo del segmento.

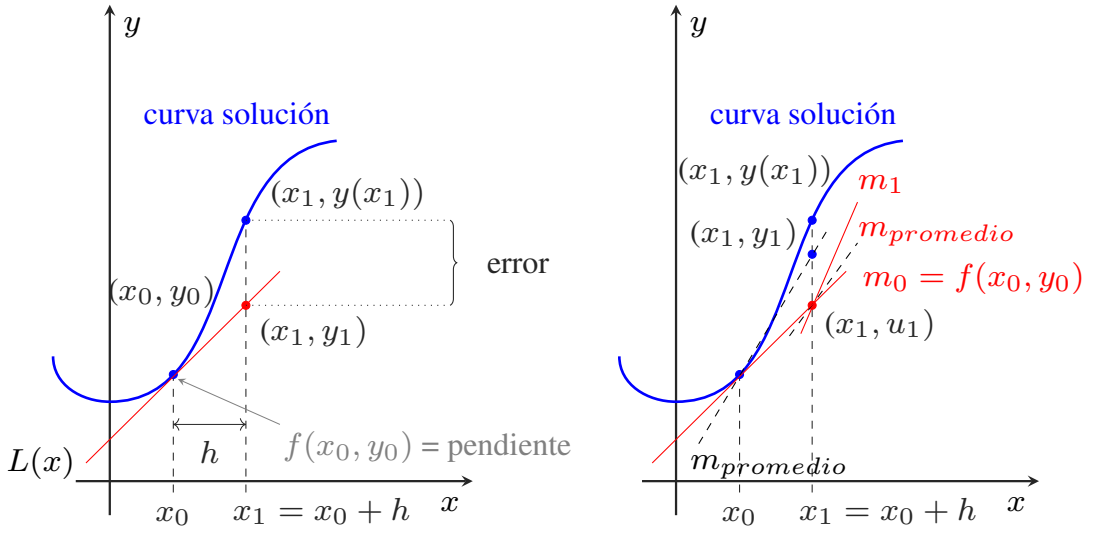
La ecuación de la recta tangente de $y = y(x)$ en el punto (x_0, y_0) será:

$$L(x) = y_0 + f(x_0, y_0)(x - x_0) \tag{9}$$

Si ahora obtenemos un pequeño incremento positivo h del eje x y sustituyendo x por $x_1 = x_0 + h$ en (9) tendremos:

$$L(x_1) = y_0 + f(x_0, y_0)(x_0 + h - x_0) \tag{10}$$

o lo que es lo mismo:



(a) Método de Euler. *(b) Método de Euler mejorado.*

Figura IV.2: La recta tangente nos ofrece una aproximación del punto $(x_1, y(x_1))$.

$$y_1 = y_0 + f(x_0, y_0)h \tag{11}$$

donde $y_1 = L(x_1)$. Un detalle importante a tener en cuenta será el cálculo de la segunda y las posteriores rectas tangentes, pues no serán "tangentes" como tal, ya que (x_1, y_1) está sobre la primera tangente y no sobre el punto $(x_1, y(x_1))$. Este hecho producirá un error –error de discretización– que se irá acumulando conforme vayamos obteniendo los siguientes puntos e incluso podría separarse tanto de la aproximación que haría inútil su cálculo. Otro error que siempre estará presente es el error de redondeo, aunque se desprecia.

De forma general, las formulas de iteración para el método de Euler son las siguientes:

$$\begin{cases} x_{n+1} = x_n + h \\ y_{n+1} = y_n + f(x_n, y_n)h \end{cases} \tag{12}$$

3.2. El método de Euler mejorado

La mejora en el método de Euler se produce al realizar un promedio de pendientes para intentar de esta manera aproximarnos aun más a la solución. Para ello se realiza un primer "tanteo" calculando con ayuda del método de Euler (sin mejora) una primera estimación para después corregir ese valor (con la mejora).

Las ecuaciones finales para la versión mejorada son las siguientes:

$$\begin{cases} x_{n+1} = x_n + h \\ u_{n+1} = y_n + f(x_n, y_n)h \\ y_{n+1} = y_n + \dfrac{f(x_n, y_n) + f(x_{n+1}, u_{n+1})}{2} h \end{cases} \tag{13}$$

Hay que ver, que se ha realizado una sustitución de y_{n+1} en el lado derecho de la ecuación (pues aparecería en ambos lados y no se podría despejar). La solución es cálcular y_{n+1} con la aproximación de Euler ($y_{n+1} = y_n + f(x_n, y_n)h$) vista en (12) que será u_{n+1} y sustituirla. Para x_{n+1} es fácil: sería sólo añadir nuestro tamaño de paso fijo.

3.3. Runge-Kutta

El método de Runge-Kutta de cuarto orden[2], abreviado como **RK4**, es más preciso que los anteriores métodos vistos con el fin de obtener soluciones (aproximadas) para problemas con valores iniciales.

Ahora serán necesarias cuatro evaluaciones de $f(x, y)$ en cada paso. Al igual que en el método de Euler mejorado, se requiere el cálculo de una pendiente promedio ponderando pendientes para cada segmento. Esto es,

$$y_{n+1} = y_n + h(w_1 k_1 + w_2 k_2 + ... + w_m k_m) \tag{14}$$

donde m es el orden del método.

Las fórmulas completas son las siguientes:

$$\begin{cases} x_{n+1} = x_n + h \\ k_1 = f\big(x_n, y_n\big) \\ k_2 = f\big(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_1\big) \\ k_3 = f\big(x_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_2\big) \\ k_4 = f\big(x_n + h, y_n + hk_3\big) \\ y_{n+1} = y_n + \dfrac{h}{6}\big(k_1 + 2k_2 + 2k_3 + k_4\big) \end{cases} \tag{15}$$

4. El atractor de Lorenz

Un sistema conocido de ecuaciones diferenciales ordinarias es el atractor de Lorenz determinado por el siguientes sistema:

[2]Conocido como el método clásico de Runge-Kutta, pues hay otros métodos con diferentes órdenes.

$$\begin{cases} \dot{X} = -\sigma X + \sigma Y, \\ \dot{Y} = -XZ + rX - Y, \\ \dot{Z} = XY - bZ \end{cases} \tag{16}$$

En física, por honor a Newton se suele emplear un tipo de notación llamada "Notación de Newton", cuando la ecuacion diferencial tiene una sola variable independiente, y ésta es el tiempo, o en otras palabras cuando X es función del tiempo, se utiliza "el punto". Hubiera sido equivalente a utilizar:

$$\begin{cases} \dfrac{dX}{dt} = -\sigma X + \sigma Y, \\ \dfrac{dY}{dt} = -XZ + rX - Y, \\ \dfrac{dZ}{dt} = XY - bZ \end{cases} \tag{17}$$

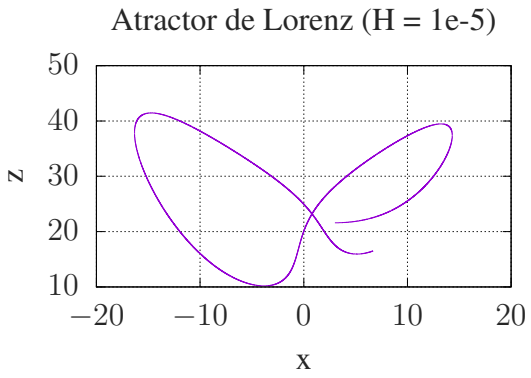
En la Figura IV.3 se puede ver la proyección $x - z$ obteniendo así el famoso *efecto mariposa* que es el el resultado del cálculo de 150.000 puntos $(x(t), z(t))$ aplicando el método de Euler o tambien llamado el método de Runge-Kutta de primer orden. Cuanto menor sea nuestro tamaño de paso h mejor será la aproximación; sin embargo, requerirá más pasos –una mayor computación– para llegar al valor t. El método de Euler no es el más recomendable debido a la acumulación del error pero sí el más sencillo. Sin embargo se verá como Cython puede intervenir para utilizar un paso de tamaño a medida sin que por ello repercuta en el tiempo del cálculo excesivamente.

5. Código

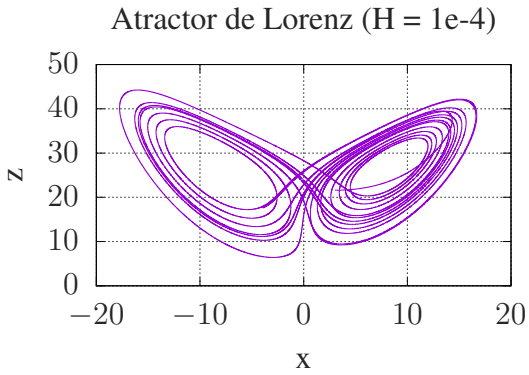
A continuación se muestra el código de Python que luego se convertirá en Cython. El original resuelve el plano $x - z$ de la ecuación diferencial de Lorenz a través del método de Euler. Este método no es el más recomendable, pues el error generado por la discretización se irá acumulando a lo largo de las siguientes iteraciones. Sin embargo esto se puede contrarrestar si se aplica un tamaño de paso h pequeño.

5.1. Python puro

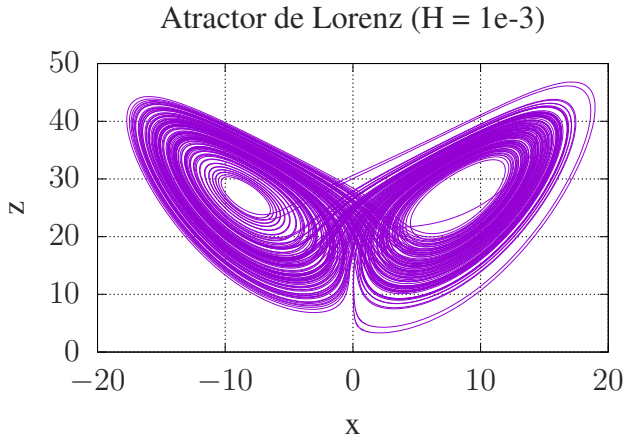
A continuación se muestra la función para la resolución del atractor de Lorenz en Python puro.



(a) Mejor precisión, mayor coste de computación.



(b) Precisión intermedia.



(c) Peor precisión, menor coste de computación y solución deseada.

Figura IV.3: Las tres gráficas contienen el mismo número de puntos, sin embargo, se ha variado el tamaño de paso.

Código IV.1: Función en Python puro para la resolución del sistema de Lorenz.

```
1  def lorenz(tiempo_fin):
2      SIGMA = 10.0
3      R = 28.0
4      B = 8/3.0
5      H = 1e-7
6      h_incremento = H
7      x = 3.0
8      y = 19.2
9      z = 21.6
10
11     for i in range(tiempo_fin):
12         x = x + H*(SIGMA*y - SIGMA*x)
13         y = y + H*(-x*z+R*x - y)
14         z = z + H*(x*y - B*z)
```

```
15          h_incremento += H
```

5.2. Cython con tipos, directivas del compilador y librerias externas

Para la función de `lorenz` se han tipado todas las constantes requeridas por el sistema de ecuaciones como se muestra en el Código IV.2. Se ha tipado para las constantes a tipo `double` así como el tamaño del paso y las variables `x`, `y` y `z`. Se ha tipado a tipo **`int`** el índice de la variable `i` de la iteración.

Código IV.2: Función en Cython para la resolución del sistema de Lorenz.

```
1  def lorenz(puntos):
2      cdef double SIGMA = 10.0
3      cdef double R = 28.0
4      cdef double B = 8/3.0
5      cdef double H = 1e-7
6
7      cdef double h_incremento = H
8      cdef double x = 3.0
9      cdef double y = 19.2
10     cdef double z = 21.6
11
12     cdef int i
13     for i in range(puntos):
14         x = x + H*(SIGMA*y - SIGMA*x)
15         y = y + H*(-x*z+R*x - y)
16         z = z + H*(x*y - B*z)
17         h_incremento += H
```

6. Cálculo de tiempos

Para este cálculo de tiempos al igual que en el algoritmo de Bellman-Ford y en Ford-Fulkerson analizados anteriormente se ha utilizado el mismo equipo, esto es, un procesador *Intel® Core™ i7-8700K CPU @ 3.70GHzx12* con la distribución GNU/Linux Ubuntu 18.04.3 LTS.

Se adjunta en la Figura IV.4 los tiempos generados para el código de Python y Cython (incluyendo la compilación directa a través del código de Python puro) del atractor de Lorenz. La resolución numérica ha sido utilizando el método de Euler.

Se ha hecho uso del comando `timeit` de Python en su versión 3.6 con el fin de obtener una precisa medida de los tiempos. Un ejemplo de su uso se puede ver en el Código IV.3

Código IV.3: Ejemplo de la obtención de tiempos con el módulo timeit en el cálculo del sistema de Lorenz.

```
1  $python3.6 -m timeit --unit sec -s "import cython_lorenz" "cython_lorenz.
       generar_datos(175e6)"
2  10 loops, best of 3: 0.0333 sec per loop
```

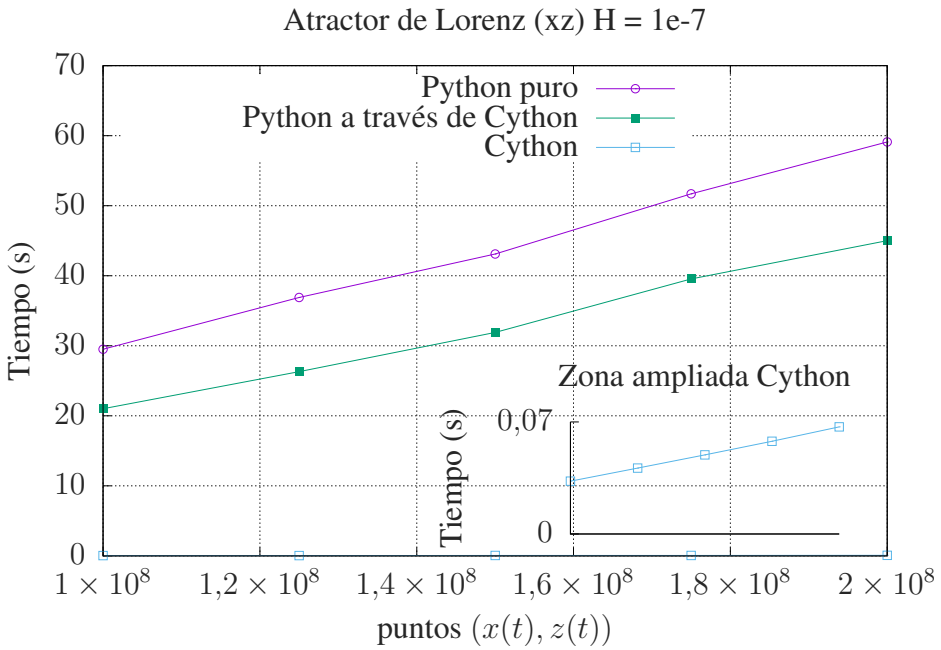


Figura IV.4: Cálculo del sistema de Lorenz. La escala de Cython ha sido ampliada. Hay una aceleración de Cython de hasta 800 veces respecto a Python puro.

7. Conclusiones

Este capítulo ha girado entorno a las ecuaciones diferenciales, su clasificación, el uso de campo de pendientes, los métodos más comunes para su resolución y finalmente se ha hecho uso de Cython en el sistema Lorenz para verificar si puede ser una buena herramienta para obtener soluciones muy aproximadas sin que por ello repercuta en el tiempo de computación.

Se ha comprobado que con Cython podemos, ya sea para su representación o para su cálculo y de forma acelerada y sencilla, calcular una trayectoria de Lorenz con valores iniciales $x(0) = 3,0$, $y(0) = 19,2$, $z(0) = 21,6$ y $0 \leq t \leq 2e - 8$.

Debido a que la resolución de EDOs implica métodos iterativos, se podrá de forma sencilla acelerar el cálculo, independientemente del método numérico que se utilice, y lograr unos muy buenos resultados.

Capítulo V

Conclusiones generales

A lo largo de este documento se ha visto una breve introducción al lenguaje Cython, su compilación (haciendo uso del compilador *gcc*) y la utilización de perfiles para el análisis del código. Se ha trabajado en la implementación de diferentes algoritmos en sus distintas versiones (Python puro, Python a través del compilador de Cython y exclusivamente mediante Cython). Este proceso ha sido gradual, aplicando lo aprendido durante la realización de este trabajo, y siempre con el objetivo de conseguir reducir los tiempos a través del uso de Cython.

Los dos primeros algoritmos en los que se ha usado Cython se hallan en el campo de la programación dinámica. Estos algoritmos han sido *Bellman-Ford* y *Ford-Fulkerson*. En *Bellman-Ford* se preveía una notable mejora cuando se comenzó a transformar el código a su version de Cython ya que, con la técnica de relajamiento del algoritmo y sus tres bucles anidados, se pasó de 51 minutos a casi 2 minutos en el tiempo empleado de computación.

Con el algoritmo de *Ford-Fulkerson* se ha tenido la oportunidad de usar librerías externas en C, más concretamente, el uso de colas, lo que en el entorno de pruebas aumentó la aceleración de forma considerable respecto a su versión de Python puro. En los resultados finales se ha apreciado una aceleración de aproximadamente 7 veces respecto a la versión en Python. Es posible que debido a las complejidades internas del propio algoritmo, como es el poco uso de bucles anidados e índices, no se hayan obtenido los resultados que se esperaban desde un principio.

Se han visto tres métodos para la resolución numérica de ecuaciones diferenciales. Estos han sido el método de Euler (junto con su versión mejorada) y el método de Runge-Kutta. Los tres hacen uso de un valor dado para proporcionarnos el valor siguiente lo que se traduce en un proceso iterativo de un bucle. Es por esto, que aplicar Cython será sencillo y a la vez se obtendrá una mejora inmediata respecto a la versión Python pura. En la resolución del sistema de Lorenz, usando el método de Euler, se obtuvo una aceleración de unas 800 veces tipando únicamente las variables junto con el índice del bucle. Se pasó de 1 minuto a aproximadamente unos 70 ms.

Apéndices

Apéndice A

Entrevista a Gregory C. Ewing

First of all, thank you Greg for your time in this difficult situation.

Q: How did your interest in programming and then compilers begin?

A: For as long as I can remember I've been interested in machines generally, finding out what's inside them and how they work. I'm not sure when I became interested in computers particularly, but it was somewhere between ages 5 and 13. I read all the library books I could find about computers and programming. I think my interest in high-level languages began there, since I remember writing a few programs in FORTRAN and COBOL, although I had no chance of being able to run them.

I learned about computing in a very bottom-up way, starting with transistors and logic gates. In my teens I built a few home-brew 8-bit micros, extremely simple machines by today's standards. They taught me about low-level computer architectures and machine languages.

The first high level language I got to use hands-on was BASIC on my school's Apple II. My home machine at the time had a BASIC ROM available for it, but it was a horrible dialect, so I wrote my own BASIC interpreter. (Written on scrap paper, hand assembled, entered using a hex monitor, and saved on cassette tape. I must have had a lot more patience in those days!)

During that time I took an evening course in Pascal programming at a polytechnic. This led me to write what was probably my first compiler, in BASIC on the Apple II. It took a vaguely Pascal-like language and translated it into BASIC.

I attempted to write a Pascal compiler for my home machine, but my ideas on how to write a parser were rather hazy back then, and it never got very far.

At university, I found all aspects of computer science interesting, but I was particularly fascinated by the whole subject of languages and their implementations. Compilers were complex and somewhat mysterious things, and I wanted to learn more about how they worked.

I've dabbled with designing and implementing languages quite a lot over the years, but I think Pyrex was my first really successful compiler project. It's certainly the first one that got serious use by anyone else!

Q: What advice would you give to students who want to break into this field.

A: That's very hard for me to answer. I got where I am now by following my nose, pursuing whatever I was interested in as best I could with what was available to me. Things are so different now that the route I followed doesn't really apply any more.

One thing I can say is that if you really have no idea where to start, then you can't go far wrong by starting with Python. It's by far the most beginner-friendly language I know of, and the Internet provides a wealth of information for learning about it, and any other aspect of computing you want to know about.

Another thing I can say is that the only way to learn to program is to do it. Lots of it. Same as with amytning else, really. That may sound like a lot of work, but if you truly have a passion for the subject, it won't seem like work.

Q: Which branches of mathematics do you think are important to become a good programmer or what particularly benefited you?

A: I don't think you need much mathematical knowledge at all for programming. I do think that if you're the kind of person who is interested in maths and likes solving mathematical puzzles, then you probably have the kind of brain needed to be good at programming.

Some areas of computer science theory might require some maths background, e.g. a familiarity with log and exponential functions will be helpful when studying algorithmic complexity, and some probability theory for compression and cryptography. But programming itself doesn't require anything more than basic arithmetic and the ability to think logically.

Q: To create Pyrex you had in-depth knowledge of the Python/C API. How long did it take you to develop a first version of Pyrex?

A: I don't really remember. Maybe a couple of months? I already knew quite a lot about the internals of Python from other things I'd done.

Q: What was one of the biggest challenges you came across while developing it?

A: I don't remember any single part of it being particularly more challenging than any other part. I had a big "aha" moment right at the beginning when I figured out what it should be like, and then it was just a matter of putting in the work to make it happen.

The biggest difficulty was probably the sheer size and complexity of it. It grew into something much more elaborate than I had anticipated, and it was all one big pile of intertwined stuff that I had to keep in my head. If I'd developed it any further I would probably have had to reorganise it to decouple things more.

Q: The "cloc" tool launched about 20,000 lines of code. Did you ever think that it would end up being so important?

A: I never thought about how big it might get in terms of total lines of code. I did get a bit alarmed one day when I noticed I had a single source file with 5000 lines in it, though!

I never imagined it would become the basis for such an influential project. That's not the kind of thing you can plan for.

Thank you very much again, Greg. Take good care of yourself.

You're welcome. You take care too.

Apéndice B

Entrevista a Stefan Behnel

Puede acceder a esta entrevista desde el blog[1] personal de Stefan Behnel.

First of all, thank you Stefan for your time in this difficult situation.

Thanks for asking me.

Q: How did your interest in programming and then compilers begin?

A: I have a pretty straight forward background and education in computer science and software development. But I'm not a compiler expert. In fact, I'm not even working on a compiler in the true sense. I'm working on Cython, which is a source code translator and code generator. The actual native code generation is then left to a C compiler. However, we avoid that distinction ourselves in the project because in the end, people use Cython to compile Python down to native code. So the distinction is more of an implementation detail.

I came to Cython through a bit of a diversion. I needed a Python XML library for the proof-of-concept implementation of my doctor's thesis somewhere around 2005. Not so long before that, Martijn Faassen had started writing an ElementTree-like wrapper for the XML library libxml2, called lxml, which had several features that I needed and was easy enough for me to hack on to get the features implemented that I was missing.

lxml was written in a code generator called Pyrex, and I ended up implementing a couple of features in that code generator that helped me in my work on lxml. Not all of these changes were accepted upstream, at least not in a timely fashion, and at some point I found that others had that problem, too, and had ended up with their own long-term forks. Together with Robert Bradshaw and William Stein from the University of Washington in Seattle, USA, we decided to fork Pyrex for good, and start a new official project, which we named Cython. That was in 2007, and I've worked on the Cython project ever since.

Q: What advice would you give to students who want to break into this field?

A: Read code. Seriously. There is a lot that you can learn at a university about algorithms, about smart ideas that people came up with, about ways to tell and decide what's smart and what isn't, about the way things work (and should work) in general. A CS de-

[1] `http://blog.behnel.de/posts/my-responses-to-a-cython-dev-interview.html`

gree is an excellent way to set a basis for your future software design endeavours.

But there's nothing that comes close to reading other people's code when you're trying to understand how things work in real life and why the tools at hand don't do what you want them to do. And then fixing them to do it.

Q:Which branches of mathematics do you think are important to become a good programmer or what particularly benefited you in Cython optimisation, for example?

A: I would love to say that my math education at university helped me here and there, but in retrospect, I need to admit that I could have come to the same point where I stand now with just my math lessons at school (although those were pretty decent, I guess). I would claim that statistics are surprisingly important in real life and software development, and are not always handled deeply enough at school (nor in CS studies), IMHO. Even just the understanding that the result of a benchmark run is just a single value in a cloud of scattered results really helps putting those numbers in context in your head.

There are definitely fields in software development in which math is more helpful than in the fields I've touched mostly. Graphics comes to mind, for example. But I think what's much more important than a math education is the ability to read and learn, and to be curious of the work of others. Because these days, 95+ % of our software development work is based on what others have already done before us (and for us). Use existing tools, learn how they work and what their limits are, and then extend those limits when you need to.

Q: If I'm not mistaken, since April 2019 you have also been a core developer in CPython: what responsibilities does this position entail?

A: The main (and most obvious) difference is the ability to click the green merge button on github. :) Seriously, you can do a lot of great work in a project without ever clicking that button. You can create tickets, investigate bugs, write documentation, advertise cool projects to others, help people use them, participate in design discussions, write feature PRs. You can move a project truly forward without being a "core developer". But once you have the merge right, you are taking over the responsibility for the code that you merge by clicking that button, wherever that code came from. If that code breaks someone's computer at the end of the world, you are the one who has to fix it, somehow. Even just by reverting the merge, but you have to do something.

Being a core developer in a project is really more of an obligation than an honour. But it can also give you a better standing in a project, because others can see that you are taking responsibility for it. So it comes with a bit of a social status, too.

Q: Cython has been and is a key tool in scientific projects, such as the Event Horizon Telescope. Which scientific libraries are you missing in Cython right now? Are

there any special ones that you are working on?

A: I'm not working on scientific libraries myself, although I know a lot of people from other projects in the field. I'm not missing anything here. :)

OTOH, I like hearing about things that others do with Cython. And I like to help others to make Cython do great things for them.

The really cool thing about OpenSource software development is that I'm creating "eternal" values every day. Whatever I write today may end up helping some person on the other side of the planet (or next door) to invent something cool, to answer the last questions about life, the universe and everything, to save the world or someone else's life. That's their projects, their ideas and their work, but it's the software that I am writing together with lots of other people that helps them get their work done. And that is a great feeling.

Q: Are there any important features that you would like to implement in Cython in the future?

A: The issue tracker has more than 740 open tickets. :) But that answer misses the point. I think the most important goal is to keep helping users getting unstuck when they run into something that they can't really (or easily) solve themselves. Cython is a tool for others to use for their own needs. It should continue to achieve that.

Q: How do you see Cython ten years from now?

A: I never liked that question when interviewing dev candidates, and I'm not going to answer it now. ;-) Ten years is about an eighth of a human's total lifetime. And it's half of an eternity in tech. That's a *very* long time. I like how Albert Einstein put it: "predictions are hard to make, especially about the future".

Thank you very much again for your time, Stefan. Take good care of yourself.

Have fun and stay safe.

Apéndice C

Código para la generación de un profile

Aquí se muestra el contenido del script que genera un profile para un módulo cualquiera.

```
1  import pstats, cProfile
2  import aquí_el_modulo_a_utilizar
3
4  cProfile.runctx("aquí_el_modulo_a_utilizar", globals(), locals(), "Profile.prof")
5  s = pstats.Stats("Profile.prof")
6  s.strip_dirs().sort_stats("time").print_stats()
```

Apéndice D

Código Python para la generación de matrices y cálculo de tiempos

Aquí se presenta el código de las **dos clases** utilizadas tanto en el algoritmo **Bellman-Ford** cómo en **Ford-Fulkerson**.

La primera clase es `SerializarMatrices`, que genera matrices aleatorias (del tipo `MatrizAdyacencia`) de tamaños varios junto a diferentes propiedades según su `factor_densidad` y el tamaño máximo de peso `max_peso`. El primer método de esta clase es `generar_and_serializar_aleatorias`, que genera las matrices, las guarda en formato binario y anota su ruta donde han sido guardadas en un fichero. El otro método es `obtener_ficheros_serializados`, que devuelve un lista de rutas de las matrices serializadas guardadas.

El código de la clase `SerializarMatrices` es el siguiente:

```
1  class SerializarMatrices:
2      @staticmethod
3      def generar_and_serializar_aleatorias(factor_densidad,
4                                            max_peso,
5                                            num_start,
6                                            num_end,
7                                            step):
8
9          file = open('save_files', 'w')
10         ma = MatrizAdyacencia()
11         while (num_start <= num_end):
12             ma.aleatoria(num_start, factor_densidad, max_peso)
13             path_matriz = "%s/%d.npy" % (Tiempos.DIRECTORIO_FILES, num_start)
14             np.save(path_matriz, ma.c_np)
15             file.write("%s\n"%path_matriz)
16             num_start += step
17         file.close()
18
19     @staticmethod
20     def obtener_ficheros_serializados():
21         file = open('save_files', 'r')
22         lines = file.readlines()
23         file.close()
24         return lines
```

La segunda clase, `Tiempos`, tiene un **método público** llamado `ejecutar_from_files` que recibe un algoritmo y una lista de rutas de las matrices serializadas. Por cada ruta de la lista, se carga esa matriz en cuestión (con uso de la función `load` de la librería `numpy`) y esa matriz en memoria se pasa al algoritmo a emplear. Después, se hará uso de la llamada **privada** `_obtener_tiempo` que ejecutará el método `ejecutar` del algoritmo. Una

vez devuelto el tiempo, se guardará en un fichero con dos columnas, una de ellas el número de vértices que contiene ese grafo junto su tiempo. De esta forma se irá iterando con todos los grafos restantes.

El código de la clase `Tiempos` es el siguiente:

```
1  class Tiempos:
2      DIRECTORIO_FILES = "matrices_files"
3      def __init__(self):
4          self.tiempos = []
5          self.numVertices = []
6          self.file = open('nodos_tiempos.txt', 'w')
7
8      def ejecutar_from_files(self, algoritmo, arr_files):
9          self.file.write("%s\t%s\n"%('Nodos', 'Tiempo'))
10         for p_file in arr_files:
11             m = np.load(p_file.rstrip())
12             algoritmo.set_obj_matriz(MatrizAdyacencia(m))
13             df = self._obtener_tiempo(algoritmo)
14             self.tiempos.append(df)
15             self.numVertices.append(algoritmo.m.get_num_vertices())
16             self.file.write("%d\t%f\n"%(algoritmo.m.get_num_vertices(), df))
17         self.file.close()
18         print("tiempos: %s", self.tiempos)
19         print("Vertices: %d", self.numVertices)
20
21     def _obtener_tiempo(self, algoritmo):
22         d0 = time.time()
23         print("Matriz con %d vertices."% algoritmo.m.get_num_vertices())
24         t = algoritmo.m.get_num_vertices()-1
25         print("| --- ejecutando...[%s]" % algoritmo.nombre)
26         algoritmo.ejecutar(0, t)
27         df = time.time()-d0
28         print("| --- Tiempo: %f" % df)
29         return df
```

Apéndice E

Código Python para el Algoritmo de Bellman-Ford

En este apéndice se muestra a continuación el código utilizado para la implementación del algoritmo de Bellman-Ford en su versión de Python puro.

El código es el siguiente:

```
class MatrizAdyacencia:
    def __init__(self, m_np = None):
        self.m = m_np
        self.dic = {}

        if m_np is not None:
            self._convert_to_dic()

    def aleatoria(self, n, factor, max_w):
        self.m = np.zeros([n, n])
        for i in range(n):
            for j in range(n):
                if i != j:
                    self.m[i][j] = np.inf
                    r = random.random()
                    if r < factor:
                        neg = 1
                        #neg = -1 if (r < 0.001) else 1
                        #self.m[i][j] = random.uniform(-1,1)*max_w
                        self.m[i][j] = np.random.rand()*max_w*neg
        self._convert_to_dic()

    def _convert_to_dic(self):
        self.dic = {}
        n = len(self.m[0])

        for i in range(n):
            lista = []
            for j in range(n):
                w = self.m[i][j]
                if w != np.inf and w != 0:
                    lista.append((j, w))
            self.dic[i] = lista

    def get_dic(self):
        return self.dic

    def get_adyacentes(self, idx_nodo):
        return self.dic.get(idx_nodo)

    def get_num_vertices(self):
        return len(self.m)
```

```
class BellmanFord:
    nombre = 'Bellman-Ford'
    def __init__(self, s, m = None):
        self.m = m
```

```
5           self.s = s
6           self.d = []
7           self.p = []
8
9       def _iniciar_relax(self):
10          num_vertices = self.m.get_num_vertices()
11          for i in range(num_vertices):
12              self.d.append(np.inf)
13              self.p.append(None)
14          self.d[self.s] = 0
15
16      def _relax(self, u, v, w):
17          u_d = self.d[u]
18          u_v = self.d[v]
19          if u_v > u_d + w:
20              self.d[v] = u_d + w
21              self.p[v] = u
22
23      def set_obj_matriz(self, m):
24          self.m = m
25
26      def ejecutar(self):
27          num_vertices = self.m.get_num_vertices()
28          self._iniciar_relax()
29          for i in range(num_vertices-1):
30              for j in range(num_vertices):
31                  for v, w in self.m.get_adyacentes(j):
32                      self._relax(j, v, w)
33
34          for i in range(num_vertices):
35              for v, w in self.m.get_adyacentes(i):
36                  if self.d[v] > self.d[i] + w:
37                      raise Exception('Ciclo negativo detectado')
```

```
1   TRABAJANDO_CON_PYTHON_PURO = False
2
3   def main():
4       print("> trabajando con...Python puro")
5       if TRABAJANDO_CON_PYTHON_PURO:
6           print("generando matrices y serializandolas...[Python puro]")
7           SerializarMatrices.generar_and_serializar_aleatorias(.85, 10, 500, 2500,
                500)
8       rutas = SerializarMatrices.obtener_ficheros_serializados()
9
10      print("ejecutando Bellman-Ford...")
11      nodo_s = 0
12      bf = BellmanFord(nodo_s)
13
14      t = Tiempos()
15      t.ejecutar_from_files(bf, rutas)
```

Apéndice F

Código Cython para el Algoritmo de Bellman-Ford

En este apéndice se expone el código completo de Cython para el algoritmo Bellman-Ford.

```
# Jonathan Ruiz Gallardo
from libc.stdlib cimport malloc, free
cimport cython
import numpy as np
cimport numpy as np
import random
import time

cdef class MatrizAdyacencia:
    cdef public np.float64_t[:,:] m
    cdef public dict dic

    def __cinit__(self, np.ndarray[np.float64_t, ndim=2] m_np = None):
        self.m = m_np
        self.dic = {}

        if m_np is not None:
            self._convert_to_dic()

    cdef aleatoria(self, int n, float factor, int max_w):
        cdef Py_ssize_t i, j
        self.m = np.zeros([n, n], dtype=np.float64)

        for i in range(n):
            for j in range(n):
                if i != j:
                    self.m[i,j] = np.inf
                    r = random.random()
                    if r < factor:
                        neg = 1
                        #neg = -1 if (r < 0.001) else 1
                        self.m[i,j] = np.random.rand()*max_w*neg

        self._convert_to_dic()

    @cython.boundscheck(False)
    @cython.wraparound(False)
    @cython.initializedcheck(False)
    cdef _convert_to_dic(self):
        self.dic = {}
        n = self.m.shape[0]

        cdef Py_ssize_t i, j
        cdef float w = 0
        for i in range(n):
            lista = []
            for j in range(n):
                w = self.m[i, j]
                if w != np.inf and w != 0:
```

```
                    lista.append((j, w))
                w = 0
            self.dic[i] = lista

    def get_dic(self):
        return self.dic
```

```
cdef class BellmanFord:
    nombre = 'Bellman-Ford'
    cdef public MatrizAdyacencia m
    cdef int s
    cdef float *d
    cdef list p

    def __cinit__(self, int s):
        self.s = s
        self.d = NULL
        self.p = []

    def set_obj_matriz(self, MatrizAdyacencia m):
        self.m = m

    @cython.boundscheck(False)
    @cython.wraparound(False)
    cdef _iniciar_relax(self):
        cdef int num_vertices = self.m.m.shape[0]
        cdef Py_ssize_t i
        for i in range(num_vertices):
            self.d[i] = np.inf
            self.p.append(None)
        self.d[self.s] = 0

    @cython.boundscheck(False)
    @cython.wraparound(False)
    cdef _relax(self, Py_ssize_t u, Py_ssize_t v, float w):
        if self.d[v] > self.d[u] + w:
            self.d[v] = self.d[u] + w
            self.p[v] = u

    @cython.boundscheck(False)
    @cython.wraparound(False)
    cdef ejecutar(self):
        cdef Py_ssize_t i, j
        cdef Py_ssize_t u, v
        cdef float w = 0
        cdef Py_ssize_t num_vertices

        self.d = <float*>malloc(self.m.m.shape[0]*sizeof(float))

        if not self.d:
            raise MemoryError()

        self._iniciar_relax()

        num_vertices = self.m.m.shape[0]
        for i in range(num_vertices-1):
            for j in range(num_vertices):
                for v, w in self.m.dic.get(j):
                    self._relax(j, v, w)

        for i in range(num_vertices):
            for v, w in self.m.dic.get(i):
                if self.d[v] > self.d[i] + w:
                    raise Exception('Ciclo negativo detectado')

```

```
59          if self.d != NULL:
60              free(self.d)
61
62          print(self.p)
63
64      def wrapper_ejecutar(self):
65          self.ejecutar()
```

```
1  TRABAJANDO_CON_PYTHON_PURO = False
2
3  def main():
4      print("> trabajando con...Cython")
5      if TRABAJANDO_CON_PYTHON_PURO:
6          print("generando matrices y serializandolas... [cython]")
7          SerializarMatrices.generar_and_serializar_aleatorias(.85, 10, 2, 3000,
               500)
8      rutas = SerializarMatrices.obtener_ficheros_serializados()
9
10     print("ejecutando Bellman-Ford...")
11     s = 0
12     bf = BellmanFord(s)
13
14     t = Tiempos()
15     t.ejecutar_from_files(bf, rutas)
```

Apéndice G

Código Python para el Algoritmo de Ford-Fulkerson

```
class MatrizAdyacencia:
    def __init__(self, c_np = None):
        self.c_np = c_np
        self.dic = {}

        if c_np is not None:
            self._convert_to_dic()

    def aleatoria(self, n, factor, max_w):
        self.c_np = np.full([n, n], np.inf)
        for i in range(n):
            for j in range(i, n):
                if i != j:
                    self.c_np[i][j] = np.inf
                    r = random.random()
                    if r < factor:
                        self.c_np[i][j] = np.random.randint(1,max_w)
                else:
                    self.c_np[i][j] = 0

        self._convert_to_dic()

    def _convert_to_dic(self):
        self.dic = {}
        n = len(self.c_np[0])
        for i in range(n):
            lista = []
            for j in range(n):
                if self.c_np[i,j] != np.inf and self.c_np[i,j] != 0:
                    lista.append(j)
            self.dic[i] = lista

    def get_num_vertices(self):
        return len(self.c_np)
```

```
class FordFulkerson:
    nombre = 'Ford-Fulkerson'
    def __init__(self, m = None):
        self.m = m

    def set_obj_matriz(self, m):
        self.m = m

    def ejecutar(self, s, t):
        m_flow = 0
        previos = [None]*self.m.get_num_vertices()
        while self.bfs(s, t, self.m, previos):
            min_cap = self._calcular_minimo(s, t, previos)
            m_flow += min_cap
            t_tmp = t
            while s != t_tmp:
                prev = previos[t_tmp]
```

```
18                 self.m.c_np[t_tmp][prev] += min_cap
19                 self.m.c_np[prev][t_tmp] -= min_cap
20                 t_tmp = prev
21
22         print("Flow maximo: {}".format(m_flow))
23
24     def _calcular_minimo(self, s, t, previos):
25         # t (target) se va sustituyendo por el nodo previo (hacia atras)
26         # de esta forma cuando t alcance el origen habremos llegado al inicio
27         # y habremos comprobado todos las aristas u -> v
28         min_cap = np.inf
29         t_tmp = t
30         while s != t_tmp:
31             min_cap = min(min_cap, self.m.c_np[previos[t_tmp]][t_tmp])
32             t_tmp = previos[t_tmp]
33         return min_cap
34
35     def bfs(self, s, t, m, previos):
36         vistos = [False]*m.get_num_vertices()
37         q = Queue()
38
39         vistos[s] = True
40         q.put(s)
41
42         while not q.empty():
43             u = q.get(0)
44             for v in m.dic.get(u):
45                 if vistos[v] == False and m.c_np[u][v] > 0:
46                     vistos[v] = True
47                     previos[v] = u
48                     q.put(v)
49         if vistos[t]:
50             return True
51         return False
```

```
1  TRABAJANDO_CON_PYTHON_PURO = False
2
3  def main():
4      print("> trabajando con...Python puro")
5      if TRABAJANDO_CON_PYTHON_PURO:
6          print("generando matrices y serializandolas... (.90, 1000, 1000, 5000,
               500)")
7          SerializarMatrices.generar_and_serializar_aleatorias(.90, 1000, 1000,
               5000, 500)
8      rutas = sm.obtener_ficheros_serializados()
9
10     print("ejecutando Bellman-Ford...")
11     nodo_s = 0
12     bf = BellmanFord(nodo_s)
13
14     t = Tiempos()
15     t.ejecutar_from_files(bf, rutas)
```

Apéndice H

Código Cython para el Algoritmo de Ford-Fulkerson

En este apéndice se expone el código completo de Cython para el algoritmo Ford-Fulkerson.

```
# distutils: sources = queue.c
# Jonathan Ruiz Gallardo
from libc.stdlib cimport malloc, free
cimport cython
cimport numpy as np
import numpy as np
import random
import time
import os, sys
cimport cqueue

cdef class MatrizAdyacencia:
    cdef public np.float_t[:,:] c_np
    cdef public dict dic

    def __cinit__(self, c_np = None):
        self.c_np = c_np
        self.dic = {}

        if c_np is not None:
            self._convert_to_dic()

    cdef aleatoria(self, int n, float factor, int max_w):
        cdef Py_ssize_t i, j
        self.c_np = np.full([n, n], np.inf, dtype=np.float)
        for i in range(n):
            for j in range(i, n):
                if i != j:
                    self.c_np[i,j] = np.inf
                    r = random.random()
                    if r < factor:
                        self.c_np[i,j] = np.random.randint(1,max_w)
                else:
                    self.c_np[i][j] = 0
        self._convert_to_dic()

    cdef _convert_to_dic(self):
        cdef Py_ssize_t i, j
        self.dic = {}
        n = len(self.c_np[0])
        for i in range(n):
            lista = []
            for j in range(n):
                if self.c_np[i,j] != np.inf and self.c_np[i,j] != 0:
                    lista.append(j)
            self.dic[i] = lista
```

```
cdef class FordFulkerson:
    nombre = 'Ford-Fulkerson'
```

```
3       cdef public MatrizAdyacencia m
4       def __cinit__(self):
5           pass
6
7       def set_obj_matriz(self, MatrizAdyacencia m):
8           self.m = m
9
10      cdef ejecutar(self, int s, int t):
11          cdef float m_flow = 0
12          cdef float min_cap = 0
13          cdef Py_ssize_t prev, t_tmp
14          cdef int *previos = <int*>malloc(self.m.c_np.shape[0]*sizeof(int))
15
16          if not previos:
17              raise MemoryError()
18
19          while self.bfs(s, t, self.m, previos):
20              min_cap = self._calcular_minimo(s, t, previos)
21              m_flow += min_cap
22              t_tmp = t
23              while s != t_tmp:
24                  prev = previos[t_tmp]
25                  self.m.c_np[t_tmp,prev] += min_cap
26                  self.m.c_np[prev,t_tmp] -= min_cap
27                  t_tmp = prev
28
29          if previos != NULL:
30              free(previos)
31          print("Flow maximo: {}".format(m_flow))
32
33      cdef _calcular_minimo(self, int s, int t, int *previos):
34          cdef Py_ssize_t t_tmp = t
35          cdef float min_cap = np.inf
36          # t (target) se va sustituyendo por el nodo previo (hacia atras)
37          # de esta forma cuando t alcance el origen habremos llegado al inicio
38          # y habremos comprobado todos las aristas u -> v
39          while s != t_tmp:
40              min_cap = min(min_cap, self.m.c_np[previos[t_tmp],t_tmp])
41              t_tmp = previos[t_tmp]
42          return min_cap
43
44      @cython.boundscheck(False)
45      @cython.wraparound(False)
46      cdef bfs(self, int s, int t, MatrizAdyacencia m, int *previos):
47          cdef Py_ssize_t i, u, v
48          cdef float w = 0.0
49          cdef int *vistos = <int*>malloc(m.c_np.shape[0]*sizeof(int))
50          cdef int esta_visto = 0
51
52          if not vistos:
53              raise MemoryError()
54
55          memset(vistos, 0, sizeof(int)*m.c_np.shape[0])
56
57          cdef cqueue.Queue *q = cqueue.queue_new()
58
59          if not q:
60              raise MemoryError()
61
62          vistos[s] = 1
63          cqueue.queue_push_tail(q, <int*>s)
64          while not cqueue.queue_is_empty(q):
65              u = <int>cqueue.queue_pop_head(q)
66
67              for v in m.dic.get(u):
68                  if vistos[v] == False and m.c_np[u][v] > 0:
```

```
69                      vistos[v] = 1
70                      previos[v] = u
71                      cqueue.queue_push_tail(q, <int*>v)
72
73          esta_visto = vistos[t]
74
75          if q != NULL:
76              cqueue.queue_free(q)
77
78          if vistos != NULL:
79              free(vistos)
80
81          if esta_visto:
82              return True
83          return False
84
85      def wrapper_ejecutar(self, int s, int t):
86          self.ejecutar(s, t)
```

```
1   TRABAJANDO_CON_PYTHON_PURO = False
2
3   def main():
4       print("> trabajando con...Cython")
5       if TRABAJANDO_CON_PYTHON_PURO:
6           print("generando matrices y serializandolas... (.90, 1000, 1000, 5000,
                500)")
7           SerializarMatrices.generar_and_serializar_aleatorias(.90, 1000, 1000,
                5000, 500)
8
9       rutas = SerializarMatrices.obtener_ficheros_serializados()
10      print(*rutas)
11      print("ejecutando FordFulkerson...")
12      bf = FordFulkerson()
13      t = Tiempos()
14      t.ejecutar_from_files(bf, rutas)
```

Apéndice I

Código C para la estructura *_typeobject* de CPython

La estructura mostrada a continuación pertenece al proyecto CPython.

```
1  struct _typeobject {
2      PyObject_VAR_HEAD
3      const char *tp_name; /* For printing, in format "<module>.<name>" */
4      Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */
5
6      /* Methods to implement standard operations */
7
8      destructor tp_dealloc;
9      Py_ssize_t tp_vectorcall_offset;
10     getattrfunc tp_getattr;
11     setattrfunc tp_setattr;
12     PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
13                                     or tp_reserved (Python 3) */
14     reprfunc tp_repr;
15
16     /* Method suites for standard classes */
17
18     PyNumberMethods *tp_as_number;
19     PySequenceMethods *tp_as_sequence;
20     PyMappingMethods *tp_as_mapping;
21
22     /* More standard operations (here for binary compatibility) */
23
24     hashfunc tp_hash;
25     ternaryfunc tp_call;
26     reprfunc tp_str;
27     getattrofunc tp_getattro;
28     setattrofunc tp_setattro;
29
30     /* Functions to access object as input/output buffer */
31     PyBufferProcs *tp_as_buffer;
32
33     /* Flags to define presence of optional/expanded features */
34     unsigned long tp_flags;
35
36     const char *tp_doc; /* Documentation string */
37
38     /* Assigned meaning in release 2.0 */
39     /* call function for all accessible objects */
40     traverseproc tp_traverse;
41
42     /* delete references to contained objects */
43     inquiry tp_clear;
44
45     /* Assigned meaning in release 2.1 */
46     /* rich comparisons */
47     richcmpfunc tp_richcompare;
48
49     /* weak reference enabler */
50     Py_ssize_t tp_weaklistoffset;
51
```

```
52      /* Iterators */
53      getiterfunc tp_iter;
54      iternextfunc tp_iternext;
55
56      /* Attribute descriptor and subclassing stuff */
57      struct PyMethodDef *tp_methods;
58      struct PyMemberDef *tp_members;
59      struct PyGetSetDef *tp_getset;
60      struct _typeobject *tp_base;
61      PyObject *tp_dict;
62      descrgetfunc tp_descr_get;
63      descrsetfunc tp_descr_set;
64      Py_ssize_t tp_dictoffset;
65      initproc tp_init;
66      allocfunc tp_alloc;
67      newfunc tp_new;
68      freefunc tp_free; /* Low-level free-memory routine */
69      inquiry tp_is_gc; /* For PyObject_IS_GC */
70      PyObject *tp_bases;
71      PyObject *tp_mro; /* method resolution order */
72      PyObject *tp_cache;
73      PyObject *tp_subclasses;
74      PyObject *tp_weaklist;
75      destructor tp_del;
76
77      /* Type attribute cache version tag. Added in version 2.6 */
78      unsigned int tp_version_tag;
79
80      destructor tp_finalize;
81      vectorcallfunc tp_vectorcall;
82  };
```

Bibliografía

[1] Cython: The Best of Both Worlds, *Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, Kurt Smith*, S Scientific Python

[2] Cython tutorial, *Stefan Behnel*, Proceedings of the 8th Python in Science Conference (SciPy 2009)

[3] Fast numerical computations with Cython, *Dag Sverre Seljebotn*, Proceedings of the 8th Python in Science Conference (SciPy 2009)

[4] Pyrex: A language for writing Python extension modules, *Gregory C. Ewing* `https://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/version/Doc/About.html`

[5] EL LENGUAJE DE PROGRAMACION C, *Brian W. Kernighan, Dennis M. Ritchie*, 2ª edición

[6] Expert C Programming Deep C Secrets, *Peter van der Linden*, 2ª edición

[7] Compiladores: principios, técnicas y herramientas, *Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman*, 2ª edición

[8] Algorithms in C: Graph algorithms, *Robert Sedgewick*, Third edition, 2002

[9] Cálculo, *Robert A. Adams*, 6ª edición

[10] ECUACIONES DIFERENCIALES con problemas con valores en la frontera, *DENNIS G. ZILL, WARREN S. WRIGHT*, 8ª edición

[11] ECUACIONES DIFERENCIALES Y PROBLEMAS CON VALORES EN LA FRONTERA, *R. Kent Nagle, Edward B. Saff, Arthur David Snider*, 4ª edición

[12] Introduction to Algorithms, *Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein*, Third edition, 2009

[13] C Algorithms, *Simon Howard*, `https://fragglet.github.io/c-algorithms/`

[14] Python/C API Reference Manual `https://docs.python.org/3/c-api/index.html`

[15] Struct object.h https://github.com/python/cpython/blob/master/Include/object.h


UAM
UNIVERSIDAD AUTONOMA
DE MADRID